Lyvathon Design Specs

Lyvathon is a statically typed Java/Python hybrid in which all operators precede their operands, and parentheses are used for all grouping (except string literals, which as in Python are delimited with single or double quotes). Lyvathon code can be embedded in a LVM text file (Lyvathon Markup language). LVM is similar to HTML, except open tags begin with a brace bracket and a keyword, and the closing tag is simply a close brace bracket. Any text enclosed in a tag is preceded by a vertical slash (|). LVM also shares features of the Wikipedia markup language, so it's simpler than HTML.

Lyvathon is both a subset and superset of Python, except a given class can only inherit from a single base class (no multiple inheritance). Lyvathon borrows the interface feature from Java, in which the interface keyword is renamed to "scool". Two other new keywords include "scoolref" (built-in function returns a scool reference) and "abclass" (abstract class). File extensions include .LV (source code), .LVA (assembler code), .LVC (compiled code), and .LVM (Lyvathon Markup language).

Keyboard Aid

This optional feature enables hyphens, open parentheses, and close parentheses to be entered by typing single quotes, commas, and periods, respectively. When enabled, keyboard aid can be temporarily suppressed by using the Ctrl key in conjunction with typing single quotes, commas, and periods (no character substitution takes place). By convention, hyphens are used to separate words in multi-word identifiers, but single quotes are easier to type than hyphens. Similarly, commas and periods are easier to type than parentheses.

Special Characters

- `()` grouping
- `-_` used in identifiers
- `;` end of stmt.
- `:` dot operator
- `" '` string delimiters
- `\` escape char.
- `#` comment
- `{* *}` block comment

Grammar Notation

- Non-terminal symbol: <symbol name>
- Optional text in brackets: [*text*]
- Repeats zero or more times: [*text*]…
- Repeats one or more times: <symbol name>…
- Pipe separates alternatives: *opt1* | *opt2*
- Comments in *italics*
- Advanced features flagged as **

Compiler and Assembler

The Lyvathon Compiler translates source code into compiled code, and optionally into assembler code. Assembler code is an intermediate language which is much simpler than source code, although both source code and assembler code are in the form of text files. During Lyvathon development, the developer uses the Lyvathon Assembler, which converts assembler code (hand-written by the developer) into compiled code. This is a necessary step enabling the LYvathon Runtime Environment (LYRE) to be tested prior to the development of the Lyvathon Compiler.

Assembler Grammar

Each token is separated from adjacent tokens with white space. Parentheses count as tokens. Consecutive parentheses need no separating white space.

<source file>:
    <module>...

<module>:
    (<mod> [<global>] [<class>]...)

<global>:
- (module ([<impmod>]...) [<def>]... [<glbdef>])

<class>:
- (<class id> ([<mod>] <base id>) ([<var>]...) [<def>]...)

<def>:
    (<id> ([<parm>]...) ([<var>]...) <block>)

<glbdef>:
    (global ([<var>]...) <block>)

<mod>:
<class id>:
<base id>:
<parm>:
<var>:
    <id> *// identifier*

<impmod>:
    <tupmod> [(<tuple>...)]

<tupmod>: *// no spaces*
    <mod>
    <mod> **:** <id>

<tuple>: *// no spaces*
    <id>
    <id> **:** <id>

<block>:
    (<hdr> [<tok>]...)

<tok>:
    <keyword>
    <expr>

<expr>:
    <functok>
    <vartok>
    <const>
    <block>
    <dot expr>

<functok>: *// no spaces*
    [[<mod> **:**] [<class id>]] **:** <id>

<vartok>: *// no spaces*
    <id> *// local var.*
    [[<mod> **:**] [<class id>]] **:** <id>

<const>:
    <num> *// integer*
    <float> *// no. with dec. pt.*
    <string lit>

<hdr>:
    <keyword>
    <lvfunc>
    <functok>

<lvfunc>:
    <id> *// built-in function*

<dot expr>:
    (dot <vartok><dotelem>...)

<dotelem>:
    <vartok>
    (<functok> [<expr>]...)

<comment>:
    # [<char>]...

    // must occur on line by itself
    // no white space before #

Lyvathon Grammar

White space occurs between tokens (parentheses and semicolon need no adjacent white space, also any semicolon before a close parenthesis may be omitted):

<source file>:
- [<line-comment>] [<vars>] [do <block>] [<dec def>]… [<class>]… [do <block>]

<import stmt>:
    import <module>
    import (<module>…)
    from <rel module> import <mod list>
    from <rel module> import all

<module>:
    <name>
    (<name> as <name>)
    (**:** <name>… [as <name>])

<mod list>:
    <id as>
    (<id as>…)

<id as>:
    <mod id>
    (<mod id> as <name>)

<mod id>:
    <mod name>
    <class name>
    <func name>
    <var name>

<rel module>:
    (<colon list> [<name>]...)
?   <name>

<class>:
- (<cls typ> <name> [<base class>] [<does>] [<vars>] do <dec def>…)
- (scool <name> [<does>] do [<const decl>]... [<dec hdr>]...)

<cls typ>:
    class
    abclass

<does>:
    does (<scool name>...)

<scool name>:
<base class>:
    <name>
    (**:** <name><name>…)

<const decl>:
    const <name><const expr> **;**

<dec hdr>:
    [<dec>]… <def hdr>

<dec def>:
    [<dec>]… <def>

<dec>:
    @ <call expr>

<def hdr>:
- proc <name> ([<var>]…)
- func <type><name> ([<var>]…)

<def>:
- proc <name> ([<var>]…) [<vars>] do <block>
- func <type><name> ([<var>]…) [<vars>] do <block>

<vars>:
    var (<var>...)

<var>:
    [static] <type><name> **;**
    [static] <type> (<name>...) **;**

<type>:
    <simple type>
    <complex type>

<simple type>:
    bool | int | long | float | double | array | dict | lisp | bitarray | string | bytezero | bytes | func

<complex type>:
    <class name>
    (<class name><method name>)

<block>:
    (<stmt-semi>…)

<stmt-semi>:
    <stmt> **;**

<stmt>:
pass
<if stmt>
<while stmt>
<for stmt>
** <try stmt>
<disruptive stmt>
<call stmt>
<asst stmt>
<del stmt>
<print stmt>
<import stmt>

<disruptive stmt>:
<continue stmt>
<break stmt>
<return stmt>
** <raise stmt>

<call expr>:
- (<name> [<expr>]…)
- (**:** <obj expr> [<colon expr>]…
  (<method name> [<expr>]…))
- (call <expr>…)

<call stmt>:
- (<name> [<expr>]…)
- **:** <obj expr> [<colon expr>]…
  (<method name> [<expr>]…)
- call <expr>…

<colon expr>:
<name>
(<name> [<expr>]…)

<asst stmt>:
<asst op><name><expr>
<asst op><target expr><expr>

<asst op>:
set | addset | minusset | mpyset | divset |
idivset | modset | shlset | shrset |
andset | orset | xorset

<target expr>:
(**:** <name> [<colon expr>]… <name>)
(slice <arr><expr> [<expr>])
(slice <arr><expr> all)

<arr>: *// string or array*
<name>
<expr>

<if stmt>:
- if <expr><block> [elif <expr><block>]…
  [else <block>]

<while stmt>:
while <expr> do <block>
do <block> while <expr>

<for stmt>:
- for <name> in <expr> do <block>
- for <name> (<expr><expr> [<expr>]) do
  <block>

<try stmt>:
- try <block> <except clause>… [else
  <block>] [finally <block>]
- try <block> finally <block>

<except clause>:
except <name> [as <name>] <block>

<raise stmt>:
raise [<expr> [from <expr>]]

<return stmt>:
return [<expr>]

<break stmt>:
break

<continue stmt>:
continue

<del stmt>:
del <expr>

<print stmt>: *// built-in func*
(print [<expr>]…)
(echo [<expr>]…)

<expr>:
<keyword const>
<literal>
<name>
(<unary op><expr>)
(<bin op><expr><expr>)
(<multi op><expr><expr>…)
(quest <expr><expr><expr>)
<array expr>
<dict expr>
<bitarray expr>
<string expr>
<bytezero expr>
<bytes expr>
<target expr>

<obj expr>
<cast>

<obj expr>:
<name>
<call stmt>

<unary op>:
minus *negate*
notbits *bitwise not*
not

<bin op>:
<arith op>
<comparison op>
<shift op>
<bitwise op>
<boolean op>

<arith op>:
div | idiv | mod | mpy | add | minus

<comparison op>:
ge | le | gt | lt | eq | ne | is | in

<shift op>:
shl | shr

<bitwise op>:
andbits | orbits | xorbits

<boolean op>:
and | or | xor

<multi op>:
mpy | add | or | and |
strdo *% operator*
strcat *+ operator*

<const expr>:
<literal>
<keyword const>

<literal>:
<num lit>
<string lit>
<bytes lit>

<array expr>:
(array [<expr>]…)

<bitarray expr>:
(bitarray <expr>)

<dict expr>:
(dict [<pair>]…)

<pair>:
(<name><expr>)
(<literal><expr>)

<cast>:
(cast <type><expr>)

<lisp funcs>:
(quote <expr>)
(compile <expr>)
(exec <expr>)

The quote *function takes a block of code and treats it like data, with an implied* list *keyword after every open parenthesis. The* compile *function compiles the output of the* quote *function, which is then executed by the* exec *function.*

No white space allowed between tokens, for rest of Lyvathon Grammar:

<white space>:
<white token>...

<white token>:
<rest-comment>
<non-line-comment>

<rest-comment>:
<non-line-comment><line-comment>

<line-comment>:
[<char>]... <new-line>

<non-line-comment>:
<white char>
<blk-comment>

<blk-comment>:
{* [<char>]... *}

<white char>:
<space> | <tab> | <new-line>

<name>:
- [<underscore>]… <letter> [<alnum>]… [<hyphen-alnum>]… [<underscore>]…

<hyphen-alnum>:
<hyphen><alnum>…

<alnum>:
<letter>
<digit>

In plain English, names begin and end with zero or more underscores. In between is a letter followed by zero or more alphanumeric characters. Names may also contain hyphens, where each hyphen is preceded and succeeded by an alphanumeric character.

<num lit>:
    <dec int>
    <oct int>
    <hex int>
    <bin int>
    <float>

<dec int>:
    0
    [<hyphen>] <any digit except 0> [<digit>]…

<float>:
    <dec int><fraction> [<exponent>]
    <dec int><exponent>

<fraction>:
    <dot> [<digit>]…

<exponent>:
    <e> [<sign>] <digit>…
<e>:
    e | E

<sign>:
    + | -

<oct int>:
    0o <octal digit>…

<hex int>:
    0x <hex digit>…
    0X <hex digit>…

<bin int>:
    0b <zero or one>…
    0B <zero or one>…

<octal digit>:
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<hex digit>:
    <digit>
    A | B | C | D | E | F
    a | b | c | d | e | f

<keyword const>:
    None
    True
    False

<colon list>:
    : [**:**]…

<string lit>:
    [<str prefix>] <short long>

<str prefix>:
    r | u | R | U

<short long>:
    ' [<short item>]... '
    " [<short item>]... "
    ' ' ' [<long item>]... ' ' '
    " " " [<long item>]... " " "

<short item>:
    <short char>
    <escaped str char>

<long item>:
    <long char>
    <escaped str char>

<short char>:
    any source char. except "\", newline, or
    end quote

<long char>:
    any source char. except "\"

<bytes lit>:
    <byte prefix><shortb longb>

<byte prefix>: *// any case/order*
    b | br

<shortb longb>:
    ' [<shortb item>]... '
    " [<shortb item>]... "
    ' ' ' [<longb item>]... ' ' '
    " " " [<longb item>]... " " "

<shortb item>:
    <shortb char>
    <escaped char>

<longb item>:
    <longb char>
    <escaped char>

<shortb char>:
    any ASCII char. except "\", newline, or
    end quote

<longb char>:
    any ASCII char. except "\"

<escaped char>:
\newline
ignore "\", newline chars.
\\ *backslash*
\" *double quote*
\' *single quote*
\a *bell*
\b *backspace*
\f *formfeed*
\n *new line*
\r *carriage return*
\t *tab*
\v *vertical tab*
\ooo *octal value = ooo*
\xhh *hex value = hh*

<escaped str char>:
<escaped char>
\N{name} *Unicode char. = name*
\uxxxx *hex value (16-bit) = xxxx*
\Uxxxxxxxx
hex value (32-bit) = xxxxxxxx

Semicolon Grammar

The semicolon switch (the default) allows statements normally enclosed in parentheses to instead be terminated with semicolons, with no need for whitespace surrounding semicolons (similar to parentheses). What follows assumes the switch is off, and statements are enclosed in parentheses.

<import stmt>:
(import <module>)
(import (<module>…))
(from <rel module> import <mod list>)
(from <rel module> import all)

<def>:
- (proc <name> ([<var>]…) [<vars>] do <stmt>...)
- (func <type><name> ([<var>]…) [<vars>] do <stmt>...)

<block>:
(<stmt>…)

<raise stmt>:
(raise [<expr> [from <expr>]])

<return stmt>:
return
(return <expr>)

<if stmt>:
- (if <expr><block> [elif <expr><block>]… [else <block>])

<while stmt>:
(while <expr> do <stmt>…)
(do <block> while <expr>)

<for stmt>:
- (for <name> in <expr> do <stmt>…)
- (for <name> (<expr><expr> [<expr>]) do <stmt>…)

<try stmt>:
- (try <block> <except clause>… [else <block>] [finally <block>])
- (try <block> finally <block>)

<call stmt>:
- (<name> [<expr>]…)
- (**:** <obj expr> [<colon expr>]… (<method name> [<expr>]…))
- (call <expr>…)

<asst stmt>:
(<asst op><name><expr>)
(<asst op><target expr><expr>)

<del stmt>:
(del <expr>)

<print stmt>:
(print [<expr>]…)
(echo [<expr>]…)

Keyword Switch

When this switch is disabled (enabled by default), many keywords such as {gt, le, set} are instead abbreviated using non-alphanumeric characters: {>, <=, =}.

Lyvathon Data Structures

1. Node Size = 12 bytes
2. Node List Size = 256 nodes/page x 12 B/node = 3072 B/page
3. Page List Size = 512 page slots x 11 B/slot = 5632 bytes
    1. Bottom level Node List (or Page) has 256 Nodes
    2. Page List or Chapter has 512 slots, each slot points to a Page
    3. Slot = 4-byte ptr. + 2 x 24-bit ptrs. + updated byte
    4. 4-byte ptr. points to node list (page) = null when page is swapped out
    5. 2 x 24-bit ptrs. point to next/prev. pages in linked list
4. Chapter List Size = 2048 page lists x 4 B/page list = 8192 bytes
5. Address Space = 32 (4-byte ptr.) + 4 (16 bytes/node) = 36 bits = 64 GB
6. Page Addr: 4-bit book no + 11-bit chapter no + 9-bit page idx = 24 bits
7. Addr Value: 24-bit page addr + 8-bit node idx = 32 bits
8. Node Types:
    1. Object Ref Node: 0 bit, 31-bit refcount, 0 bit, 31-bit root (attr) ptr., code ptr.
    2. Object Value Node: 0 bit, 31-bit refcount, 1 bit, header, 4-byte value
    3. Lisp Node: 1 bit, 31-bit refcount, 2 x 4-byte ptrs. to lisp, obj ref, obj val nodes
    4. Tree Node: 2 x 16-bit hdrs., 2 x 4-byte values
    5. Stack Node: 4-byte hdr., 4-byte value, 4-byte next
    6. Base Node: prior base ptr., root (parm/loc) ptr., next ptr.
    7. Long Node: 64-bit long, double, or bitarray
    8. Callback Node: obj ref, code ptr.
    9. String Node: 3 x 4-byte Unicode chars.
    10. ByteZero Node: 12 bytes, null-terminated
    11. Bytes Node: 12 bytes
    12. Array Node: same as stack node, used for string, bytezero, bytes, bitarray values
    13. String List: bytezero value, may contain newline chars.
    14. Dict Leaf Node: string list, array node (list of values)
9. Header Values:
    1. node, boolean, int, long, float, double, object, lisp, string, bytezero, bytes, bitarray, array, dict, dict leaf, callback, op, paren, null
10. Binary Tree: has root, max path size = 32 bits
    1. object node
    2. base node
    3. array, dict
11. Class Inheritance:
    1. Code ptr. may point to method in ancestor class
    2. Current class includes all ancestor attributes
12. Page Types:
    1. Book (up to 16 of these), Chapter, Page
    2. Also called chapter list, page list, node list, respectively
13. Swap File:
    1. 16 chapter lists (2048 ptrs. x 4 B/ptr.) = 128K
    2. 16 x 2048 page lists (512 page slots x 11 B/slot) < 192 MB
    3. 16 x 2048 x 512 node lists x 3072 B/page = 48 GB
14. Tree-balancing functionality needed
15. Little or no support for arrays
16. Linked list implemented in library using Seq class:
    1. Data structure: lisp nodes, each node points to current, rest of list
    2. Properties: first/last lisp nodes, count
    3. Methods: pop, push, append, insert, delete, getnode, getnext, getprior
    4. StackSeq class: top, count, pop, push
17. Reference counting used for garbage collection

18. PageUpdate(currIdx)
    1. // move pageList[curridx] to end of occupied page list
    2. **if** prev(currIdx) = 0 **then** firstFull = next(currIdx)
    3. **else** next(prev(currIdx)) = next(currIdx)
    4. **if** next(currIdx) = 0 **then** lastFull = prev(currIdx)
    5. **else** prev(next(currIdx)) = prev(currIdx)
    6. // append currIdx to occupied page list
    7. next(currIdx) = prev(currIdx) = 0
    8. **if** lastFull = 0 **then** firstFull = currIdx
    9. **else**
        1. next(lastFull) = currIdx
        2. prev(currIdx) = lastFull
    10. lastFull = currIdx
    11. updated(curridx) = 1
    12. return
19. PageSwapNew(newIdx)
    1. Occurs when adding a new node list, forcing an old node list to be swapped out
    2. // newIdx = addr. of new page
    3. // free page list in RAM is empty
    4. oldIdx = firstFull // head of occupied page list
    5. **if** updated(oldIdx) **then** write old node list to swap file
    6. newIdx = addr. of new page
    7. currIdx = oldIdx
    8. PageAppend(curridx, newIdx)
    9. updated(currIdx) = 1
    10. return
20. NullPointer(currIdx)
    1. Occurs when null ptr. (swapped page) encountered in page list
    2. // currIdx = idx of swapped page in page list
    3. // pageList[currIdx] is null
    4. **if** next(currIdx) = 0 and firstEmpty = currIdx **then** PageSwap(curridx)
    5. **else** PageRefresh(curridx)
    6. return
21. PageRefresh(curridx)
    1. Read node list (of currIdx) from swap file
    2. newIdx = addr. of page just read
    3. PageAppend(curridx, newIdx)
    4. return
22. PageSwap(curridx)
    1. // free page list in RAM is empty
    2. oldIdx = firstFull // head of occupied page list
    3. **if** updated(oldIdx) **then** write old node list to swap file
    4. Read node list (of currIdx) from swap file
    5. Overwrite old node list with data read (addr = oldIdx)
    6. PageAppend(curridx, oldIdx)
    7. return
23. PageAppend(currIdx, newIdx)
    1. pageList[currIdx] = newIdx
    2. // remove currIdx from empty page list
    3. **if** prev(currIdx) = 0 **then** firstEmpty = next(currIdx)
    4. **else** next(prev(currIdx)) = next(currIdx)
    5. **if** next(currIdx) = 0 **then** lastEmpty = prev(currIdx)
    6. **else** prev(next(currIdx)) = prev(currIdx)

7. // append currIdx to occupied page list
8. next(currIdx) = prev(currIdx) = 0
9. **if** lastFull = 0 **then** firstFull = currIdx
10. **else**
    1. next(lastFull) = currIdx
    2. prev(currIdx) = lastFull
11. lastFull = currIdx
12. updated(currIdx) = 0

Memory Usage

- 1 K-node = 4 pages x 256 nodes/page = 1024 nodes
- Memory/user (lo) = 256 pages/user x 3072 B/page = 0.75 MB/user = 768K/user
- Memory/user (hi) = 1024 pages/user x 3072 B/page = 3 MB/user
- Memory/server = 1 GB minimum
- Users/server (lo) = 1024 MB/server x 0.33 users/MB = 341 users/server
- Users/server (hi) = 1024 MB/server x 1.33 users/MB = 1365 users/server
- Disk space/server = 1024 GB
- Disk space/user (lo) = 1024 GB/server x (0.75/1024 GB/user) = 0.75 GB/user/server
- Disk space/user (hi) = 1024 GB/server x (3/1024 GB/user) = 3 GB/user/server
- Nodes/user (lo) = 0.75 MB/user x (1/16 nodes/B) = 48 K-nodes/user = 192 pages
- Nodes/user (hi) = 3 MB/user x (1/16 nodes/B) = 192 K-nodes/user = 768 pages
- Slots/day = 2 slots/hour x 24 hours/day = 48 slots/day
- User-slots/server/day = 341 users/server x 48 slots/day = 16,384
- Users/subscriber (lo) = 100
- Users/subscriber (hi) = 200
- Subscribers/server (lo) = 16,384 user-slots/server/day x (0.01 subscribers/user) = 164
- Subscribers/server (hi) = 16,384 user-slots/server/day x (0.005 subscribers/user) = 82
- Revenue/server/year (lo) = 82 subscribers/server x $20/subscriber/year = $1640/year
- Revenue/server/year (hi) = 164 subscribers/server x $20/subscriber/year = $3280/year

Code Execution

All Lyvathon 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
```