Jovelyst Implementation

[Go Back]

Jovelyst 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
```

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 #

Data in RAM

This section DEPRECATED !!!

All Jovelyst data is stored in 256-byte pagelets or 4K array/dictionary pages. All pages (except array pages) have 16 pagelets. All user data is stored in a file on disk of up to 4 GB in size, since available RAM is probably much less than 4 GB.

To resolve a 32-bit data address, the first byte indexes the root table of up to 256 addresses. Each address in the root table points to a block table of 256 addresses. The second byte in the data address indexes the block table. The indexed address points to a 64K block. The first nybble of the third byte in the data address points to the 4K page in the block. The second nybble of the third byte in the data address points to the 256-byte pagelet in the block. The fourth byte in the data address indexes the final data location within the pagelet. For array pages the least-significant 12 bits in the data address indexes a particular array element contained in that page.

Every 64K block in RAM contains a list of 16 page headers, and each page header is of size 8 bytes. This list replaces pagelet 0 of page 0 (page 0 is never an array page). The page header contains 2 bits: swapped-out and modified. If the page is swapped out, then the rest of the page header contains 20 bits pointing to the corresponding page in the 4 GB file on disk. If the page is not swapped out, then the rest of the page header contains 2 partial data addresses, each of size 20 bits. These partial data addresses point to the next and previous pages in RAM (whether or not the corresponding page is part of the free-page list). Whenever a page in RAM is accessed (read from or written to), it is moved to the head of this doubly linked list. Whenever a page in RAM needs to be swapped out, it is selected from the tail of the doubly linked list.

Node Headers

Every 8-byte node or 4-byte data value is preceded by a 2-byte header. An 8-byte node usually consists of an address and a 4-byte data value, which itself may be an address. Some 8-byte nodes contain a 64-bit data value: a double or a long. The most significant bit of the header indicates that either the next 8 bytes are a node or the next 4 bytes are a data/address value. The second bit of the header indicates that the node/value is empty. The third bit (if needed) is used for garbage collection. A 5-bit portion of the header indicates type: boolean, char, int, long, float, double, object, lisp, string, bytezero, bytes, bitarray, array, dict, callback, op, paren, indirect, null.

Garbage Collection

A simple mark-sweep algorithm is used for garbage collection. It is probably unnecessary to make use of reference counting. The end-user may experience periodic delays whenever garbage collection takes place.

End of section DEPRECATED !!!

Code Execution

All Jovelyst 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
```

Data Structures

1. Node Size = 12 bytes **Data Structures section DEPRECATED !!!**
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