**Mike Hahn** – hahnbytes@gmail.com

# Jispoteach

Jispoteach is a distance-learning website which is based on an open source language called Jispotalk. Jispotalk (along with Bracetagger, a markup language) is used to develop web-based apps used by the tutors and students to interact with each other. Tutors must pay 15 percent of the fees they charge their students to Jispoteach: 5 percent goes to the developers of the web-based apps, 5 percent goes to the educators who develop the curriculum, and 5 percent goes to Jispoteach.com (the House).

## Typewriter Font

All body text rendered to the browsers of end-users in every Jispoteach app is in a mono-spaced, typewriter-style font. Every character takes up 2 square cells: an upper cell and a lower cell. Superscripts and subscripts are handled by employing a vertical offset of one square cell. Header text is also mono-spaced, and each character takes up 2 oversized square cells.

## Additional Formatting

The grid of characters can be subdivided into panels, which can themselves be subdivided into more panels, and so on. Any panel can contain zero or more text boxes, which may overlap each other. Vertical grid lines each take up one square cell per row of square cells. Horizontal grid lines are displayed in the same pixel row as underscore characters. Any row of square cells containing a horizontal grid line which is 2 pixels wide is taller by exactly one pixel. The following bracket characters: ( ) [ ] { } can be oriented vertically or horizontally, taking up a single column or row of at least 2 square cells, respectively. Widgets such as check boxes, radio buttons, and combo box arrows take up 4 square cells (2 by 2). Images, animations, and diagrams are contained in canvas objects, which can appear anywhere panels can appear.

## Rich-Text Mode

Eventually Jispoteach apps will support rich-text mode, in which a given header or paragraph of body text can consist of a single variable-width font. Paragraphs have before/after spacing, left/right indent, and line spacing (single, double, 1.5, etc.). Panels have margins on all 4 sides. Beginner app writers start off with mono-space mode, and then advance to rich-text mode. In both rich-text and mono-space modes, text is rendered to the HTML5 canvas object. Some features like form fields and submit buttons use normal HTML.

## Grading of Content

Users can assign a letter grade (on a 4.0 scale) to the apps/tutors they use: A = excellent, B = good, C = average, D = poor, F = fail. Users are warned prominently if a given app/tutor received any F grades in the past year. If a given user has given out more than one F grade to multiple apps/tutors in the past 12 months, then those grades are flagged as possibly unreliable.

## Page Hierarchy

Since Jispoteach uses Bracetagger instead of HTML, that Bracetagger code cannot be discovered by Google users doing searches. To get around that limitation of Jispoteach, a hierarchy of limited HTML pages exists. Each page has an HTML title, a description containing one or more HTML paragraphs, a heading, an optional short one-line description, an optional image (with an optional caption), and zero or more user-defined field-value pairs. All HTML pages contain the same set of field-value pairs. If the value is a null string then the corresponding field name is not displayed. The user can click on Up, Down, Next, Previous to navigate the tree, as well as clicking on the heading to display the corresponding Bracetagger web page.

## Scalability

All Jispotalk data is stored in 256-byte pagelets or 4K array pages. All pages (except array pages) have 16 pagelets. All data of a given user is stored in the same 4 GB file on disk, and that file may contain data for more than one user. As many 4 GB files as can fit on one server may exist, per server.

Any one server can have up to 4 GB of RAM devoted to Jispotalk data encompassing one or more users. To resolve a 32-bit data address for a given user, the first byte indexes the user root table of up to 256 addresses. Each address in the user root table points to a user block table of 256 addresses. The second byte in the data address indexes the user 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.

## Bracetagger Format

| | |
|---|---|
| Simple Tag | `{<taghdr>}` |
| Container Tag | `{<taghdr> \| <body>}` |
| Null Tag | `{\| <body>}` |
| List Tag | `{<taghdr> <list>}` |
| Body List Tag | `{<taghdr> <bodlst>}` |
| <bodlst> | `[ \| <body>]*` |
| <list> | `[<col>]*` |
| <col> | `\| [<fldval>]* \| <body>` |
| <taghdr> | `<tagname> [<fldval>]*` |
| <fldval> | `fldname=value;` |
| | `fldname;` |
| <body> | body text |
| Escape Char. | backslash (\) |
| Repetition: | `[ XYZ ]*` |

- *XYZ* repeats zero or more times

Tag Names:

- table, row, box, grid, borders
- canvas, select, quote, rt, mono
- super, sub, text, pre, br, hr, img, a, ch, p
- input, radio, checkbox, meta, jpt
- styles, include, h1..h5, b, i, u, ol, ul, jy

Mono Mode:

- body color fcolor (c/f)
- panel c/f width height borders-tag
- borders
    - left right top bottom inner outer none wleft wright wtop wbottom winner wouter
    - left=80ff44 right=0FF0000
    - wleft=0..2 wtop=0..2 (pixels)
- table c/f width height left top borders-tag
    - w/h/l/t (cell sqhttp://treenimation.net/hahnbytes/index.htmluares, not pixels)
- row c/f height borders-tag
    - c/f width colspan rowspan borders-tag
- box c/f width height left top borders-tag
- grid c/f width height left top rowcount colcount rowheight colwidth borders-tag
- gridrow c/f height borders-tag
    - c/f width borders-tag
- canvas c/f width height left top borders-tag
    - w/h/l/t (pixels)
- just=0..2 (left/center/right)
- valign=0..2 (top/center/bottom)
- jy h=0..2 v=0..2
- select
    - jpt expr (int or bool)
    - 1st, 2nd, 3rd, ...
- jpt: Jispotalk code

Fields (old):

- width=50/0.5 (pixels/ratio)
- pad=50/0.5
- x, y = 50/0.5
- height = n (pixels)
- topb=1 (pixels)
- bottomb, leftb, rightb, midb = 1
- color=FF00FF (rgb)
- fcolor=00FF00 (text)
- bcolor=000000 (borders)
- colspan, rowspan = n
- just="L/C/R"
- b, i, u (bold, italics, underline)
- coldefs
- rows, cols = n (grid size)
- id="mynode", id="mytag"
- onclick="mypage.html"

Tags RT/Mono:

- current and child panels use Rich-Text mode, not monospace
- mono overrides rt
- only in version 2

Mono Mode (cont'd):

- a
    - href="mypage.bt"
    - href="#label"
    - href="somepage.bt#label"
    - name="label"
    - onclick="(myfunc a b c)"
- img src width height left top alt borders-tag
    - src="myicon.jpeg"
- ch name/unicode
    - Sigma
    - epsilon
    - nbsp
    - x03c7

## Implementation Steps

1. Learn to use GitHub with Java project
2. Create Jabbler: multi-player Scrabble (single-user)
    1. Write Jabbler pseudo-code - ***done!***
    2. Register Jabbler project with GitHub
    3. Implement Jabbler command-line version - ***done!***
    4. Add robot player(s)
    5. Two human players use mouse/keyboard - *ignore*
3. Implement Jispotalk Assembler
4. Implement JispoTalk Runtime Environment (JTRE)
5. Implement Jispotalk Compiler
6. Learn how to develop web applications using Java
7. Implement converter: Jispotalk-to-JavaScript
8. Use Jispotalk as server-side scripting language
9. Convert Jabbler from Java to Jispotalk
10. Make Jabbler multi-player
11. Implement Jispotalk editor in Java
12. Implement Bracetagger editor in Java
13. Convert code editors from Java to Jispotalk
14. Expand Jispotalk libraries: widgets, graphics, games, math, etc.
15. Design website
16. Implement website
17. Go live
18. Purchase Google AdWords advertising
19. Implement billing
20. Implement Rich-Text mode
21. Add Rich-Text mode to Bracetagger editor

## About Me

I am Mike Hahn, the founder of Jispoteach.com. I was previously employed at Brooklyn Computer Systems as a Delphi Programmer and a Technical Writer (I worked there between 1996 and 2013). At the end of 2014 I quit my job as a volunteer tutor at Fred Victor on Tuesday afternoons, where for 5 years I taught math, computers, and literacy. I'm now a volunteer computer tutor at West Neighbourhood House. My hobbies are reading quora.com questions/answers and the news at cbc.ca. About twice a year I get together with my sister Cathy who lives in Victoria. She comes here or I go out there usually in the summer. At those times I'm much more active, but most of the year I tend to lie on the couch a lot, and not be very active. I do, however, visit my brother Dave once a month or so and I also visit my friends Main and Steph once or twice a month.

## Contact Info

Mike Hahn
Founder, Jispoteach.com
515-2495 Dundas St. West
Toronto, ON  M6P 1X4

Country: Canada
Phone: 416-533-4417
Email: hahnbytes (AT) gmail (DOT) com
Web: www.hahnbytes.com

# Jispotalk

Jispotalk is a Python dialect 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). Jispotalk code can be embedded in a Bracetagger text file. Bracetagger 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 (|). File extensions include .JPT (source code), .JPA (assembler code), .JPC (compiled code), and .BT (Bracetagger).

## Keyboard Aid

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

## Special Characters

- `( )` grouping
- `-_` used in identifiers
- `-;` end of stmt.
- `:` dot operator
- `" '` string delimiters
- `\` escape char.
- `#` comment
- `{* *}` block comment
- `{ }` Bracetagger code
- `\*}` treated as Bracetagger code
  not block comment

## Differences from Python

- Parentheses, not whitespace
- Integration with Bracetagger
- Operators come before their operands
- Single, not multiple inheritance
- Adds interfaces ("scool" defs.)
- Drops iterators and generators
- Adds lambdas
- Adds quote and list-compile functions, treating code as data

## 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 Jispotalk 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 Jispotalk development, the developer uses the Jispotalk Assembler, which converts assembler code (hand-written by the developer) into compiled code. This is a necessary step enabling the JispoTalk Runtime Environment (JTRE) to be tested prior to the development of the Jispotalk Compiler.

## Jispotalk Grammar

*White space occurs between tokens (parentheses need no adjacent white space, also any semicolon/hyphen 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>:
  def <name> ( [<var>]… )

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

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

<var>:
  <id>

<block>:
  ( [<stmt-semi>]… [<stmt>] )

<stmt-semi>:
  <stmt><endchar>
  <bt tag>

<endchar>: *// must be followed by white space*
  **; | -**

<jump stmt>:
  <continue stmt>
  <break stmt>
  <return stmt>

<pjump stmt>:
  return <expr>
  ** <raise stmt>

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

```
<stmt>:
      <open stmt>
      <closed stmt>

<open stmt>:
      <if stmt>
      <while stmt>
      <for stmt>
      ** <try stmt>
      <pjump stmt>
      <pcall stmt>
      <asst stmt>
      <del stmt>
      <import stmt>

<closed stmt>:
      <jump stmt>
      <call stmt>
      <print stmt>
      <bt tag>

<bt tag>:
      { ... }

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

<call stmt>:
•     ( <name> [<expr>]… )

<pcall stmt>:
•     : <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>

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

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

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

<for stmt>:
      for <name> in <expr> do <block>

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

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

<return stmt>:
      return

<break stmt>:
      break

<continue stmt>:
      continue

<del stmt>:
      del <expr>

<paren stmt>:
      ( <open stmt> )
      <closed stmt>

<qblock>:
      ( quote [<paren stmt>]... )
```

```
<expr>:
    <keyword const>
    <literal>
    <name>
    ( <unary op><expr> )
    ( <bin op><expr><expr> )
    ( <multi op><expr><expr>… )
    ( quest <expr><expr><expr> )
    <lambda>
    ( quote <expr>... )
    <array expr>
    <dict expr>
    <bitarray expr>
    <string expr>
    <bytezero expr>
    <bytes expr>
    <target expr>
    <obj expr>
    <cast>

<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> )

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

<lambda>:
    ( lambda ( [<id>]... ) <expr> )
    ( lambda ( [<id>]... ) do <block> )
    ( lambda ( [<id>]... ) do <qblock> )
    // must pass qblock thru compile func
```

*No white space allowed between tokens, for rest of Jispotalk Grammar (text omitted for brevity).*