Nojotalk Table of Contents

by Mike Hahn

[Go Back]

Note to the reader:

This document contains the documentation of Nojotalk, in its entirety,
along with the entire contents of its web site.

Introduction to Nojotalk

Nojotalk is at once a multiplayer HTML5 game portal and an open source tool for creating multiplayer HTML5 games and game engines. All games and game engines are written in the Nojotalk scripting language, which is interpreted on the developer's local machine, and deployed as JavaScript code on the server, running on the Node.js platform. Most games and game engines are open source and hosted for free by Nodejitsu.com (games which are very popular require game developers to pay fees to Nodejitsu). Nojotalk is implemented in Python, so it's cross-platform.

Nojotalk IDE

The Nojotalk scripting language is based on the subset of JavaScript as laid out in *JavaScript: The Good Parts*, by Douglas Crockford. However, the syntax of Nojotalk is a major departure from JavaScript, whereby infix operators are absent and parentheses are used for all grouping (except string literals are delimited with double quotes). The Nojotalk-to-JavaScript Converter is used to deploy games and game engines on the server. The NojoTalk Markup Language (NTML) is used to lay out screen elements. NTML is similar to JSON, except its syntax is a subset of Nojotalk, and its functionality is a subset of HTML. Developers have the option of writing their code using JavaScript syntax instead of Nojotalk syntax.

Nojotalk comes with 3 built-in game engines: Board (used for board games and non-game apps), CellGrid (regions where multiple players congregate), and DynaGrid (a 2-player animated game). A fourth game engine is a split screen consisting of Board and CellGrid. Game engine developers are free to modify any of the built-in game engines, or create their own game engines from scratch. Nojotalk databases are built with MongoDB, which is itself open source.

Business Model

Nojotalk has no business model: it's completely free. The Nojotalk Game Portal web site consists of a searchable directory of Nojotalk games, along with links to those games (which are hosted elsewhere). The only expenses of Nojotalk are web hosting fees of less than $300 per year.

To Do List

1. Nojotalk Byte-Code Interpreter
2. Nojotalk Compiler
3. Command Shell
4. NTML Parser
5. Converter: Nojotalk-to-JavaScript
6. Board engine
7. CellGrid engine
8. DynaGrid engine
9. Multi-user
10. Database support
11. Nojotalk IDE
12. Game Portal

Game Engines

Nojotalk comes with 3 built-in game engines: Board (used for board games and non-game apps), CellGrid (regions where multiple players congregate), and DynaGrid (a 2-player animated game). A fourth game engine is a split screen consisting of Board and CellGrid. Game engine developers are free to modify any of the built-in game engines, or create their own game engines from scratch.

DynaGrid

- Animated 2-player or single-player game
- Grid size one quarter of CellGrid (4 x 4)
- Graphics are bitmaps or solid rectangles
- Players can move in 4 directions (horizontal/vertical)
- Game objects can move in 16 directions: horizontal, vertical, diagonal, or like a chess knight
- May contain multiple levels
- Graphics objects are strings:
    - Each 16-bit character is an index into a cell array
    - Each cell is a bitmap or solid rectangle
    - Null chars. are holes in a graphics object
- Objects and players move smoothly from one grid cell to the next
- Nojotalk attempts to keep the current player in the center of the screen, by scrolling the display (which is suppressed if an empty grid row/column would otherwise be revealed)

CellGrid

- Current user's avatar in center
- Scroll grid using arrow keys
- Quarter Cell:
    - Cell contains 2 to 4 objects
    - Cell divided into 4 quarters
- User Avatar:
    - White background
    - Black text (2 uppercase letters)
    - User name: first/last name
    - Click: display user name
    - Click again: interact with user
- Non-scrolling row/column: up to 4 of these may be displayed, at the top, bottom, left or right sides of the display window
- Hyperlink:
    - Press Enter or click to follow hyperlink
    - Cell border is dotted
    - Move to a different CellGrid, or start DynaGrid/Board game
- Nojotalk attempts to keep the current player in the center of the screen, by scrolling the display (which is suppressed if an empty grid row/column would otherwise be revealed)
- Split-screen: fourth game engine consists of a CellGrid and a Board sharing the screen, divided either horizontally or vertically

Board

- Features:
    - Used by board games and most apps
    - Top-level window divided into either a grid, horizontal or vertical panels
    - Each panel contains a widget, label, stack, canvas, book, more panels, or is empty
    - Group panels similar to top-level window, containing child panels
    - Book: tabbed notebook (kind of widget)
    - Canvas contains one or more board panels (stacks)
    - A stack contains a horizontal or vertical stack of graphics objects (may have zero overlap)
- Graphics object:
    - bitmap
    - shape: rectangle, rounded rectangle, ellipse
    - text (single line)
    - group of sub-objects
- Enter key pressed:
    - fire onclick event of default button
    - insert <cr> in memo widget with input focus
    - select highlighted item in combo/list-box with input focus
- Space bar pressed:
    - insert space in text edit/memo widget
    - fire onclick event of widget with input focus
- Mouse events:
    - widget, panel, canvas, graphics object
    - double-click flag
    - right-click flag
    - modifier flags: left/right shift, ctrl, alt
    - down/up flag
    - pixel coordinates
    - callback function
- Keyboard events:
    - widget with input focus
    - virtual key code
    - modifier flags: left/right shift, ctrl, alt
    - down/up flag
    - keypress flag
    - callback function
- Joystick events:
    - direction
    - button no.
    - callback function
- Paint events:
    - widget, panel, canvas, graphics object
    - rectangle coordinates
    - callback function

Events

All users register to listen to any or all of the following events, related to a specific game object/player, a class of game objects/players (and all its subclasses), all game objects and/or all players.

- Join/leave/jump
- Move one grid cell
- Border:
    - 4 coordinates: x1, x2, y1, y2
    - 3 coordinates are non-negative
    - 4th coordinate equals -1
    - Cross flag:
        - true if object attempts to cross border
        - false if object is simply adjacent to border
- Collision (2 parties):
    - Cross flag:
        - true if object attempts to overlap other object
        - false if object is simply adjacent to other object
- Click:
    - Confirm flag is true if user's previous click event on same object occurred within a given period of time, in seconds
- Attribute change
- Timer
- Widget
- Custom

About Me

I am Mike Hahn, the founder of Nojotalk. I have been working on Nojotalk and its predecessors, in a sporadic fashion, for almost 2 decades. (In the fall of 2012 and winter/spring 2013 I made an abortive attempt at developing a database of mental health resources.) I have been employed at Brooklyn Computer Systems as a Delphi Programmer, and more recently as a Technical Writer, for almost 17 years. At the beginning of 2013 my BCS duties were greatly reduced, so I'm using Nojotalk to fill the void. I use software development to fill my days. For fun I go online and read the news, and most evenings I watch on my laptop yesterday's *The National*, a CBC TV news show. I'm also a volunteer tutor at Fred Victor on Thursday afternoons, where I teach math, computers, and literacy. Although my background is database programming, by the time Nojotalk goes live in late 2014 (or thereabouts) I hope to become an expert Python Programmer. It is my ambition, by implementing Nojotalk, to make a significant contribution in the field of web-based development tools, thereby helping developers everywhere to more easily create web-based software.

Contact Info

2495 Dundas St. West
Ste. 515
Toronto, ON M6P 1X4
Canada

Phone: 416-533-4417
Email: hahnbytes@gmail.com
Web: www.Nojotalk.com

NojoTalk Markup Language

NTML files are the Nojotalk equivalent of HTML files. Every Nojotalk project includes at least one NTML file, and all NTML files of a given project are located in the project's root directory or its subdirectories. In general, an NTML file consists of either an object or an array. An array consists of a list of zero or more values, separated by white space and enclosed in parentheses. The open parenthesis is followed by the keyword **list**. In the case of a concatenation array, the open parenthesis is followed by a plus sign (+) instead of a keyword, and all array elements are strings concatenated together.

An object consists of a list of zero or more pairs, separated by white space and enclosed in parentheses. The open parenthesis is followed by the keyword **pairs**. Each pair is enclosed in parentheses and consists of a string literal followed by a value. A value can be one of the following: an object, an array, a string literal, a number, or a constant keyword. An object having a "tag" pair equal to "script" can also have a "text" pair with a value consisting of a Nojotalk block (zero or more statements), which is always enclosed in parentheses. The value of a "type" pair is often a widget type. Examples of widget types include "button" and "checkbox". If the value of a "tag" pair is a string literal beginning with a forward slash, then that object serves to terminate the matching object without the forward slash (which lacks a "text" pair).

A constant keyword is one of the following: **true, false, null**. A string literal is delimited with single or double quotes, and uses the backslash as its escape character. Multi-line string literals are delimited with 2 double quotes ("") at both ends. Numbers can be integers or floating point. Floating point numbers may be in scientific notation. All tokens except parentheses must be separated by white space.

A top-level NTML file consists of an object containing 2 pairs: "head" and "body". The value associated with each pair is an array. One of the values in the "head" array might be an object with a "tag" pair equal to "title" and a "text" pair equal to a string literal consisting of the title of its NTML file. Another possible value in the "head" array might be a reference to a Nojotalk file. All user-defined functionality is written in the Nojotalk built-in scripting language. Here is a sample top-level NTML file:

```
(pairs
  ("head" (list
     (pairs
       ("tag" "title")
       ("text" "Sample NTML file")
     )
  ))
  ("body" (list
    (pairs
      ("tag" "h1")
      ("text" "Sample NTML file")
    )
    (pairs
      ("tag" "p")
      ("text" "Here is the body text.")
    )
  ))
)
```

Nojotalk Scripting Language

Game designers use Nojotalk as their scripting language. Nojotalk is based on the subset of JavaScript described in the book *JavaScript: The Good Parts*, by Douglas Crockford. Nojotalk differs from JavaScript only in its radically simplified syntax, in which parentheses are used for all grouping (except string literals are delimited with single or double quotes; square brackets and semicolons/commas are not used). Comments start with // or are delimited with brace brackets.

Debugger

Nojotalk programmers design, test, and debug their code locally, and then upload the output of the Nojotalk-to-JavaScript Converter to the server. When in debug mode, the Nojotalk code window lets you set breakpoints, examine variables, and single-step through your code. The debugger includes 2 command shells: the var shell and the do shell. The var shell lets you examine variables without executing any code. The do shell lets you evaluate Nojotalk expressions. The command prompts are var> and do>. To switch between shells, use the var or the do command (type var or do and then press Enter). To cycle through the current user list, press a hot key (defaults to Ctrl+U). Both Nojotalk and NTML code is entered in a similar code editor which supports syntax highlighting, smart indent, and multiple undo/redo.

Code is an Array of Tokens

Nojotalk code is represented in memory as an array of tokens (Nojotalk is implemented in Python). Every pair of parentheses in Nojotalk code corresponds to an array of tokens and/or other arrays. If the open parenthesis is followed by a token, instead of another open parenthesis, then the first element of the corresponding array contains that token in the form of a token object. If instead of a token the open parenthesis is followed by another open parenthesis, then the first element of the corresponding array contains another array. Token objects can be either keywords/operators, constants (literals), or identifiers. Each token object contains a token type (keyword/constant/identifier) and a data type (both integers). The data type is either the particular keyword/operator or data type of the constant/identifier. Token objects which are constants/identifiers also have a token value. The token value is either the value of the constant or an object reference (an object embedded in the Nojotalk function/variable data structure).

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. Similarly, commas and periods are easier to type than parentheses.

Nojotalk Syntax

- 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*

No white space allowed between tokens:

<name>:
    <letter> [<idchar>]…

<idchar>:
    <letter>
    <digit>
    <underscore>
    <hyphen> *each hyphen must be delimited with alphanumeric chars.*

<num lit>:
    <integer> [<fraction>] [<exponent>]

<integer>:
    0
    [<hyphen>] <any digit except 0> [<digit>]…

<fraction>:
    <dot> [<digit>]…

<exponent>:
    <e> [<sign>] <digit>…

<e>:
    e | E

<sign>:
    + | -

<string lit>:
    " [<esc or non-quote>]… [<non-quote>] "
    " [<esc or non-apostrophe>]… [<non-apostrophe>] "

<non-quote>:
    any Unicode character except " and \ and control character

<non-apostrophe>:
    any Unicode character except ' and \ and control character

<esc or non-quote>:
    <escaped character>
    <non-quote>

<esc or non-apostrophe>:
    <escaped character>
    <non-apostrophe>

<escaped character>:
    \" *double quote*
    \' *single quote*
    \\ *backslash*
    \/ *slash*
    \b *backspace*
    \f *formfeed*
    \n *new line*
    \r *carriage return*
    \t *tab*
    \u <4 hexadecimal digits>

<regexp lit>:
    / <regexp choice> / [g] [i] [m]

White space occurs between tokens (parentheses need no adjacent white space):

<var stmt>:
    (var <name> [<expr>])
    (var (<name or pair>…))

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

<stmt>:
    <expr stmt>
    <disruptive stmt>
    <try stmt>
    <if stmt>
    <switch stmt>
    <while stmt>
    <for stmt>
    <do stmt>

<disruptive stmt>:
    <break stmt>
    <return stmt>
    <throw stmt>

<block>:
    ([<stmt>]…)

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

<switch stmt>:
    (switch <expr><case clause>… [default <block>])

<case clause>:
    (<expr><block>)
    (case (<expr>…) <block>)

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

<for stmt>:
    (for (<initialization><condition><increment>) <block>)
    (for <name> in <expr><block>)

<initialization>:
<increment>:
    ( )
    <expr stmt>

<condition>:
    ( )
    <expr>

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

<try stmt>:
    (try <block> catch <name><block>)

<throw stmt>:
    (throw <expr>)

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

<break stmt>:
    break

<expr stmt>:
    <asst stmt>
    <invocation>
    (delete <expr><expr>)

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

<asst op>:
    = | += | -=

<refinement>:
    (**:** <expr><expr>…)

<invocation>:
    (<func name> [<expr>]…)
    ($ <expr>…)

<expr>:
    <literal>
    <name>
    (<prefix op><expr>)
    (<bin op><expr><expr>)
    (<multi op><expr><expr>…)
    (? <expr><expr><expr>)
    <invocation>
    <refinement>
    (new <invocation>)
    (delete <expr><expr>)

<prefix op>:
    typeof
    + *to number*
    - *negate*
    not

<bin op>:
    / | % | - | >= | <= | > | < | === | !==

<multi op>:
    * | + | or | and

```
<literal>:
    <num lit>
    <string lit>
    <obj lit>
    <array lit>
    <func lit>
    <regexp lit>

<obj lit>:
    ( pairs [ ( <name or string><expr> ) ]… )

<array lit>:
    ( list [<expr>]… )

<func lit>:
    ( func [<name>] <parms><body> )

<parms>:
    ( [<name>]… )

<body>:
    ( [<var stmt>]… [<stmt>]… )
```

Nojotalk Byte Code

Nojotalk source code is converted into byte code at compile-time. Every function body (as well as main, or top level) corresponds to a string of 16-bit characters. The most significant bit is the operator bit, which is zero for operators and keywords. The next most significant bit is zero for local variables and parameters, and one for global variables. The low order 14 bits for local variables and parameters encode the stack offset. The low order 14 bits for global variables/functions encode the offset value. The following JavaScript code is a sneak preview of the byte code processor, which will eventually be written in Python.

```
var stack;
var stkbase;
var funcarray;
var glbarray;
var currfunc;
var currpos;
var currbody;
var parmcount;
var varcount;

function push(val) {
    stack.push(val);
}
function pop() {
    return stack.pop();
}
function jump(addr) {
    currpos = addr;
}
function callfunc(funcidx, posidx) {
    currfunc = funcidx;
    currpos = posidx;
    funcobj = funcarray[currfunc];
    currbody = funcobj.body;
    parmcount = funcobj.parmcount;
    varcount = funcobj.varcount;
}
```

```
function mainloop() {
    stack = [];
    stkbase = 0;
    callfunc(0, 0);
    initfuncs();
    initops();
    do {
        token = currbody.charCodeAt(currpos);
        if (token < 32768) {
            operator = operators[token];
            operator();
        }
        else {
            offset = token - 32768;
            push(offset);
        }
        currpos += 1;
    }
    while (stack.length > 0);
}
function assign(addr, val) {
    addr -= 32768;
    if (addr < 16384) {
        stack[stkbase + addr] = val;
    }
    else {
        addr -= 16384;
        glbarray[addr] = val;
    }
}
function getval(addr) {
    addr -= 32768;
    if (addr < 16384) {
        val = stack[stkbase + addr];
    }
    else {
        addr -= 16384;
        val = glbarray[addr];
    }
    return val;
}
```

```
function initfuncs() {
    typeof_fn = function() {
        push(typeof(pop()));
    };
    to_number = function() {
        push(+pop());
    };
    negate = function() {
        push(-pop());
    };
    not = function() {
        push(!pop());
    };
    multiply = function() {
        push(pop() * pop());
    };
    divide = function() {
        temp = pop();
        push(pop() / temp);
    };
    remainder = function() {
        temp = pop();
        push(pop() % temp);
    };
    add = function() {
        temp = pop();
        push(pop() + pop());
    };
    subtract = function() {
        temp = pop();
        push(pop() - pop());
    };
    greater_or_equal = function() {
        temp = pop();
        push(pop() >= pop());
    };
    less_or_equal = function() {
        temp = pop();
        push(pop() <= pop());
    };
    greater = function() {
        temp = pop();
        push(pop() > pop());
    };
    less = function() {
        temp = pop();
        push(pop() < pop());
    };
    equal = function() {
        push(pop() === pop());
    };
    not_equal = function() {
        push(pop() !== pop());
    };
    refinement = function() {
        expr = pop();
        obj = pop();
        push(obj[expr]);
    };
```

```
logical_or = function() {
      addr = pop();
      flag = pop();
      if (flag) {
        push(true);
        jump(addr);
      }
};
logical_and = function() {
      addr = pop();
      flag = pop();
      if (!flag) {
        push(false);
        jump(addr);
      }
};
asst = function() {
      addr = pop();
      expr = pop();
      assign(addr, expr);
};
plus_asst = function() {
      addr = pop();
      expr = pop();
      val = getval(addr);
      assign(addr, val + expr);
};
minus_asst = function() {
      addr = pop();
      expr = pop();
      val = getval(addr);
      assign(addr, val - expr);
};
jump_local = function() {
      jump(pop());
};
branch_true = function() {
      addr = pop();
      if (pop()) {
        jump(addr);
      }
};
branch_false = function() {
      // (same as ternary)
      addr = pop();
      if (!pop()) {
        jump(addr);
      }
};
call = function() {
      addr = pop();
      push(catchaddr);
      push(stkbase);
      push(currfunc);
      push(currpos);
      callfunc(addr, 0);
};
```

```
post_call = function() {
      rtnpos = pop();
      rtnfunc = pop();
      stkbase = stack.length - parmcount
      push(varcount);
      push(rtnfunc);
      push(rtnpos);
};
return_no_val = function() {
      rtnpos = pop();
      rtnfunc = pop();
      varcount = pop();
      for (i = 1; i <= varcount; i += 1) {
            pop();
      }
      stkbase = pop();
      callfunc(rtnfunc, rtnpos);
      pop();
};
return_val = function() {
      rtnval = pop();
      rtnpos = pop();
      rtnfunc = pop();
      varcount = pop();
      for (i = 1; i <= varcount; i += 1) {
            pop();
      }
      stkbase = pop();
      push(rtnval);
      callfunc(rtnfunc, rtnpos);
      pop();
};
```

```
    array_literal = function() {
        count = pop();
        array = [];
        for (i = 1; i <= count; i += 1) {
          array.push(pop());
        };
        array.reverse();
        push(array);
    };
    object_literal = function() {
        count = pop();
        object = {};
        for (i = 1; i <= count; i += 1) {
          expr = pop();
          key = pop();
          object[key] = expr;
        };
        push(object);
    };
    throw_in_try = function() {
        jump(pop());
    };
    catch_clause = function() {
        expr = pop();
        assign(offset, expr);
    };
    throw_not_try = function() {
        expr = pop();
        do {
          rtnpos = pop();
          rtnfunc = pop();
          varcount = pop();
          for (i = 1; i <= varcount; i += 1) {
             pop();
          }
          stkbase = pop();
          callfunc(rtnfunc, rtnpos);
          catchaddr = pop();
        } while (!catchaddr);
        jump(catchaddr);
        push(expr);
    };
    call_self = function() {
        push(catchaddr);
        push(stkbase);
        push(currfunc);
        push(currpos);
        callfunc(currfunc, 0);
    };
}

function initops() {
    operators = [
        defun,
        begin,
        end,
        typeof_fn,
        to_number,
        negate,
        not,
        multiply,
        divide,
        remainder,
        add,
        subtract,
        greater_or_equal,
        less_or_equal,
        greater,
        less,
        equal,
        not_equal,
        refinement,
        logical_or,
        logical_and,
        asst,
        plus_asst,
        minus_asst,
        jump_local,
        branch_true,
        branch_false,
        call,
        post_call,
        return_no_val,
        return_val,
        array_literal,
        object_literal,
        throw_in_try,
        catch_clause,
        throw_not_try,
        call_self
    ];
}
```

Sample Byte Code

Token Name Examples:

1. Operator: **opadd**
2. Definition: **opdef myvar**
3. Variable reference: **opvar myvar**
4. Global integer constant: **opdef -123**
5. Global float constant: **opdef 3.1416E-12**
6. Global string literal: **opdef "xyz&ABC DEF_123"**

Nojotalk Code:

```
// set label to 'Hello, world!'

(= myFunction (func () (
    (= (: ($ (: document getElementById) 'myLabel') innerHTML) 'Hello, world!')
)))
```

Equivalent Byte Code:

Token:	Overhead:
op_post_call	6
loc_func1	
loc_label1	
loc_func2	
str_getElementById	
str_myLabel	
str_Hello_world	
str_innerHTML	
glb_document	
str_getElementById	
op_refinement	3
loc_func1	
op_asst	3 + 3
str_myLabel	
loc_func1	
op_call	6 + 6
loc_label1	
op_asst	3 + 3
str_Hello_world	
loc_label1	
str_innerHTML	
op_refinement	3
op_asst	3 + 4
op_return_no_val	22 .
Total:	65
Main Loop:	144 = 24 tokens x 6
Grand Total:	209 lines of JavaScript code executed