Asm-Blox User Manual

Zachary Romero

2021

Contents

1 Introduction

Asm-Blox is an attempt to make Emacs more exciting by merging it with trendy topics, namely blocks (a la Minecraft), and WASM (have like what Rust can do!). Asm-blox is an attempt to combine these elements together in a an exciting game package, taking inspiration from the budding genre of programming puzzle games.

The game tasks you with writing a program in an idealized version of WebAssembly Text to solve a simple programming puzzle. Puzzles range in difficulty and at first you'll try to get use to the unusual instructions later using what you've learned in conjunction with pre-made utility components to solve even more complex problems.

2 Installation

This package is installable under the identifier "asm-blox" on MELPA.

3 Getting Started

3.1 Puzzle Selection

By running `asm-blox` you can open the *puzzle selection page*. On this page you can see a list of all of the puzzles with their difficulty and description. You will also see the puzzles previously created on the right side of the screen. Pressing `RET` when the point is on any one of them will load said puzzle.

3.2 Saved Puzzles and Backups

Puzzles are saved in the directory indicated by the variable `asm-blox-save-directory-name`. Puzzles are saved as they are edited (ie 12 boxes with text in them). In the case that the file is saved in a bad format it may become corrupted. In this case there are backups saved each time the file is ran which end in `.backup.txt`. When a puzzle is completed, a file with the extension `.win.txt` is saved.

4 Gameplay

4.1 Overview

The gameboard consists of three rows and four columns in which the user will enter their solution. Each box will be referred to as a *code cell*. Code cells can contain WAT or a module. The following is an example of a WAT cell:

```
+--------------------+
|(left)(ne (get 0)   |
|        (const ?<)) |
|(ne(get 0)          |
|   (const ?>))(and) |
|(block(block        |
|  (br_if 0)         |
|  (send down)(down) |
|  (br 1))           |
| (send right)       |
| (right))           |
|(clr)               |
|                    |
+--------------------+
```

In later sections, I will go over the meanings of the various commands. Arrows entering the box are the box's *input ports*, The arrows goign out are the box's *output ports.* On the edges of the gameboard you will see some labeled arrows. The input of the puzzle will come through these ports. Labeled arrows going out of a box is where you have to send your answer.

The following is an example of a module cell:

```
+--------------------+
|(module heap        |
| :size 100          |
| :set-port up       |
| :seek-port left    |
| :peek-port up      |
| :offset-port left) |
|                    |
|                    |
|                    |
|                    |
+--------------------+
```

4.2 Execution

By pressing `C-c C-c` (`asm-blox-start-execution`) you can execute the current puzzle. This opens up a new buffer where the puzzle's inputs and outputs will become visible. The following keybindings will then be available to you in an execution buffer:

- `n` (`asm-blox-execution-next-command`) will perform one step of execution.
- `N` (`asm-blox--execution-next-multiple-commands`) will perform the number of steps defined by the variable. `asm-blox-multi-step-ct`.
- `r` (`asm-blox--execution-run`) will run the program until it finishes, an error occurs, or is stopped.
- `q` (`quit-window`) will stop execution.

Submitting the wrong answer will cause execution to stop as well as any runtime error.

5 WAT Command Set

The following is a complete description of all the available WAT commands in the game. **Note that the commands are case insensitive**

5.1 Execution Model of WAT Cells

Cells run on a stack-based architecture. Each command will consume zero to two items from the stack and push zero or one item. Each cell has a stack size of four. Any element past that will cause a *stack-overflow error*. If the stack doesn't have enough items, a *stack-underflow errors* occurs.

Many commands can have nested operations. These are indicated by a ... in the following sections. When forms are nested, they will run before the parent item pushing their items onto the stack before the parent command is executed. For example, the following instruction executes as would be expected in Lisp (`(+ 10 (- 20))`):

```
(add (const 10) (neg (const 20)))
```

5.2 Stack Commands

- `const` *number*: Pushes *number* onto the stack. Note that numbers may be in the form of a ASCII character (ex. `?a` for the character a)
- `set` *stack-offset*: Sets the stack item at *stack-offset* to the value at the top of the stack, popping it.
- `inc` *stack-offset* ...: Increment the value on the stack at *stack-offset*. **NOTE:** To increment the value at the top of the stack use `(inc -1)`.
- `dec` *stack-offset*: Decrement the value of the stack at *stack-offset*. **NOTE:** To decrement the value at the top of the stack use `(dec -1)`.
- `clr`: Clear the entire stack.
- `dup`: Duplicate the stack. Note: this will overflow the stack if there is more than two items on the stack.
- `drop` ...: Pop the item off the top of the stack.

5.3 Numeric Commands

- `add` . . . : Remove the top two items on the stack, add them, and push the result on the stack.
- `sub` . . . : Remove the top two items on the stack, subtract the upper item from the lower one and add the item back on the stack.
- `mul` . . . : Remove the top two items on the stack, multiply them and push the result on the stack.
- `div` . . . : Remove the top two items on the stack, divide the lower item by the upper item and push the *quotient* on the stack.
- `rem` . . . : Remove the top two items on the stack, divide the lower item by the upper item and push the *remainder* on the stack.
- `neg` . . . : Remove the top item from the stack and invert its sign.

5.4 Boolean Operations

asm-blox has no notion of true or false. The number 0 is used for false and any other number is true.

- `and` . . . : Remove the top two items on the stack, pushing 1 if they are both true, 0 otherwise.
- `or` . . . : Remove the top two items on the stack, pushing 1 if either of them is true, 0 otherwise.
- `not` . . . : Remove the top item from the stack and push its inverted truth value onto the stack.

5.5 Comparison Operations

Note that all comparison operations compare from bottom to top of the stack. This means that an operation such as `(lt (const 1) (const 3))` will be true.

- `lt` . . . : Pop two items on stack and push 1 if the bottom item is *less than* the top, 0 otherwise.
- `gt` . . . : Pop two items on stack and push 1 if the bottom item is *greater than* the top, 0 otherwise.

- `ge` . . . : Pop two items on stack and push 1 if the bottom item is *greater than or equal to* the top, 0 otherwise.
- `le` . . . : Pop two items on stack and push 1 if the bottom item is *less than or equal to* the top, 0 otherwise.
- `eq` . . . : Pop two items on stack and push 1 if the bottom item is *equal* the top, 0 otherwise.
- `ne` . . . : Pop two items on stack and push 1 if the bottom item is *not equal to* the top, 0 otherwise.
- `lz` . . . : Pop one items on stack and push 1 if it is less than zero, 0 otherwise.
- `gz` . . . : Pop one items on stack and push 1 if it is greater than zero, 0 otherwise.

5.6 Other Operations

- `nop`: Do nothing

5.7 Port Operations

The following commands are used to interact with the port network. Note that commands like `(up)` and `(down)` are given for convenience.

- `send` *port* . . . : Send the item off the top of the stack to *port* if it empty. If the port is full, block.
- `get` *port*: Push the item from *port* onto the stack.
- `up`: Push the item from the *up* port onto the stack.
- `down`: Push the item from the *down* port onto the stack.
- `left`: Push the item from the *left* port onto the stack.
- `right`: Push the item from the *right* port onto the stack.

5.8 Blocks and Loops

WAT cells come with two methods of control-flow: `loop` and `block`. If you're familiar with WAT the logic works similar.

The commands `br` and `br_if` are the two commands to work with `block` and `loop`. A `br` command will either skip to the end of a `block` or loop to the top of a `loop`. A `br` must specify which block or loop it is referring to via a number. Consider the following example:

```
(block   ; 2
 (block  ; 1
  (block ; 0
   (br <block ID>))))
```

The <block ID> above can be either 0, 1, or 2 since it is contained in three nested blocks. If <block ID> was set to 1, then control flow would jump past the middle block. If <block ID> was 2 then the control would pass all of the blocks.

Let's consider another example with `loop`. Suppose we want to send the numbers from 0 to 10 to the down port. We could write the following code:

```
(const 0) ; 1
(loop     ; 2
 (send down (get 0)) ; 3
 (set 0 (add (get 0)
            (const 1))) ; 4
 (ne (get 0) (const 10)) ; 5
 (br_if 0))              ; 6
```

1. Initialize the top of the stack to 0.
2. Setup a loop
3. Send the value at the bottom of the stack down.
4. Set the value at the bottom of the stack to be 1 plus its current value.
5. Push 1 if the item at the bottom of the stack is not equal to zero
6. If true (ie 1) is on the top of the stack, jump to the loop.

`block`, `loop`, `br` and `br_if` can be combined to create a wide variety of constructs.

Commands:

- `block`: Setup a block. Any `br` command pointing to this block will jump past the end of the block.
- `loop`: Setup a block. Any `br` command pointing to this block will jump to the beginning of this block.
- `br`: Unconditionally jump to a block.
- `br_if`: Consume the top item of the stack, jumping if true, continuing if false.

6 Module Blocks

Module cells can be constructed in addition to WAT blocks to add premade processes which add essential functionality to solve some problems. The current available types of module blocks includes: Stack, Heap, and Controller.

Module blocks are written using a s-expression definition. The spec is as follows: (module *module-kind …specification*).

6.1 Stack Cells

When the kind of a Module Cell is "stack", a stack is created. A stack reads data from an input source, adding it to an internal data store, and makes it available to an external port. The following is an example of a stack.

```
+-------------------+
|(module stack      |
| :input-port down  |
| :output-port right |
| :size 20)         |
|                   |
|                   |
|                   |
|                   |
|                   |
|                   |
+-------------------+
```

The following are spec properties of a stack module:

- `:input-port`: any value sent here will be added to the top of the stack.
- `:input-ports`: multiple input port specifications. Can not be used with `input-port`.
- `:output-port`: if another cell reads from here, it will be popped off the stack.
- `:size`: the maximum amount of elements that can be on the stack before a stack-overflow error occurs. The maximum size of a stack is 999.
- `:size-port`: the port to which the current amount of elements on the stack is written to.

6.2 Heap Cells

Heap cells are created by creating a module with `heap` as its kind. Heaps provide you with an array of memory for your cells to read and write to. The following is an example of a heap cell:

```
+--------------------+
|(module heap        |
| :read-port down    |
| :write-port down   |
| :seek-port left    |
| :offset-port left  |
| :set-port up       |
| :peek-port up)     |
|                    |
|                    |
|                    |
+--------------------+
```

- `:read-port` (out): from this port you can read the value at the current address. By reading this value, the current address will increase by one. Note that end of file is indicated by the value -999.

- `:write-port` (in): writing a value to this port will store a value at the current address. After writing a value, the current address will increase by one.
- `:offset-port` (out): from this port you can read the current offset into memory.
- `:seek-port` (in): by writing a value to this port, you can set the offset into memory.
- `:peek-port` (out): this port will have the data at the current offset. Reading from this port will not move the offset.
- `:set-port` (in): this port will set the data at the current offset. Writing to this port will not move the offset
- `:size`: You may configure the size of the heap with this variable. The maximum size allowed is 999.

6.3 Controller Cells and Editor Problems

Some problems require the special Asm-blox Editor integration. These problems, when executed, will show an editor with the expected output under it. Your goal in these problems is to get the editor to match the target text. You can interact with the editor via Controller cells. A controller cell is a Module cell with a value of "controller" for the `kind` position. By configuring the ports on the Controller cell you can move the point and insert text, sort of like how you would in Emacs itself. You can have multiple controllers in a game.

The following is an example of a controller cell:

```
+--------------------+
|(module controller  |
| :input-port left   |
| :set-point-port    |
|    down            |
| :point-port up)    |
|                    |
|                    |
|                    |
|                    |
|                    |
+--------------------+
```

Controller cells allow you to set the following properties on the `spec`:

- `:input-port`: Data sent to this input point will be written to the buffer at the current point. ASCII characters with a value of 32 through 126 can be printed. You may also send ASCII 10 (?) for a newline, 8 (?) or -1 for a backwards delete and -2 for a forwards delete.
- `:set-point-port`: Data sent to this input port will set the position of the port. A value past the bounds of the buffer will bring the point to the beginning (position 1) or the end of the buffer.
- `:char-at-port`: This output port will contain the character in front of the point.
- `:point-port`: This output port will contain the current point position.

6.4 Legacy YAML-defined Modules

In previous versions of the `asm-blox` software, modules were able to be created via YAML statements. Currently this is not recommended and may be removed in future versions of `asm-blox`.

The following is an example of a YAML block:

```
+--------------------+
|apiVersion: v1      |
|kind: Heap          |
|spec:               |
|  size: 100         |
|  setPort: up       |
|  seekPort: left    |
|  peekPort: up      |
|  offsetPort: left  |
|                    |
|                    |
|                    |
|                    |
+--------------------+
```

A YAML block has three top level items: `apiVersion`, `kind`, and `spec`. `kind` must be either `Stack`, `Heap`, or `Controller`. The spec is the same stated above but with the keys in snake case (ex. `:seek-port` becomes `seekPort`).

7 Editing Commands

The following commands may be helpful when editing asm-blox code:

- `asm-blox-start-execution` (bound `C-c C-c`): compile the code cells and create an execution buffer.
- `asm-blox-move-beginning-of-line` (bound `C-a`): Move the point to the beginning of a code cell line if in a cell, to the beginning of the line otherwise.
- `asm-blox-move-end-of-line` (bound `C-e`): Move the point to the end of a code cell line if in a cell, to the end of the buffers line otherwise.
- `asm-blox-beginning-of-buffer` (bound `M-<`): Move the point to the end of a code cell if in a cell, to the end of the buffer otherwise.
- `asm-blox-end-of-buffer` (bound `M->`): Move the point to the end of a code cell if in a cell, to the end of the buffer otherwise.
- `asm-blox-next-cell` (bound `<tab>`): Move the point to the end of the next code cell.
- `asm-blox-prev-cell` (bound `<backtab>`): Move the point to the end of the previous code cell.

7.1 Undo and Redo

Asm-blox support undo and redo on a per-cell basis.

- `asm-blox-undo` (bound `s-z`): undo a previous action in the current code cell.
- `asm-blox-redo` (bound `s-y`): redo a previous undo in the current code cell.

7.2 Advanced editing

- `asm-blox-shift-box-up` (bound `<s-up>`): Swap the current code cell with the one above the current one.
- `asm-blox-shift-box-down` (bound `<s-down>`): Swap the current code cell with the one below the current one.

- `asm-blox-shift-box-left` (bound `<s-left>`): Swap the current code cell with the one to the left of the current one.
- `asm-blox-shift-box-right` (bound `<s-right>`): Swap the current code cell with the one to the right of the current one.
- `asm-blox-kill-region` (bound `<C-w>`): kill the highlighted region of the current code cell. Note that the two ends of the region must be in the same code cell.
- `asm-blox-copy-region` (bound `<M-w>`): copy the highlighted region of the current code cell. Note that the two ends of the region must be in the same code cell.
- `asm-blox-yank` (bound `C-y`); paste the yanked region to the current code cell.

8 Customizations

The following customization options exist

- `asm-blox-save-directory-name`: The name of the directory in which all puzzles will be saved. Note that you can effectively start from scratch, clearing your progress, by setting this to a new, empty directory.