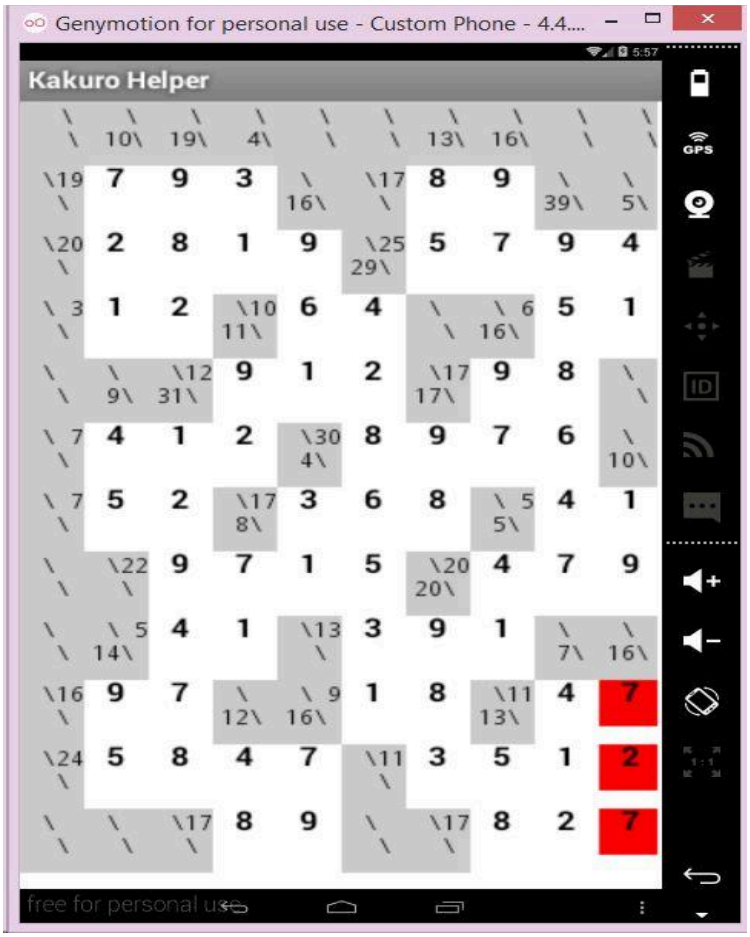
Kakuro Helper V01

Background

The New York Sunday News carries a Hidato Puzzle in its comic section. This app is meant to help solve the puzzle. This app lets you solve the puzzle, with Undo/Redo capability and error checking. Because of its size, special care needs to be taken in the Blocks Editor to keep everything collapsed, and to use a device or GenyMotion for testing. It's too big for the MIT Ai2 emulator.

Sample Puzzle

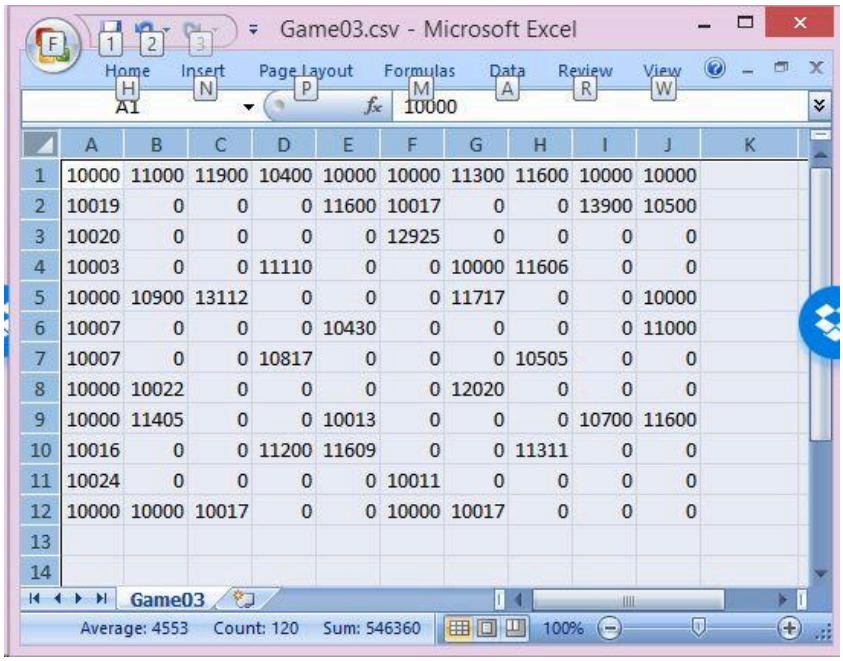
Kakuro is played on a 12 row by 10 column grid of black and white cells. The black cells contain target sums for the white cells immediately to their right and below them. The white cells start out empty, for the players to fill with numbers in the range 1 to 9. No duplicates are allowed in any vertical or horizontal range.



This sample puzzle has almost totally been solved, except for the vertical range of red cells in the lower right, which though it adds up to 16, contains duplicate 7's.

In the original puzzles, the black cells are drawn with white number sums on a black background, separated by a white diagonal line.

This implementation uses labels for the black cells, with black letters on a light grey background for readability. The diagonal line is simulated by two '\' characters, spaces, a line feed \n character, and a monospace font that just fits two lines per cell.



Input Format

Because of the grid nature of this game, it's easy to enter puzzles through a spreadsheet program. However, the black cells pose a problem: how to keep two numbers in one cell?

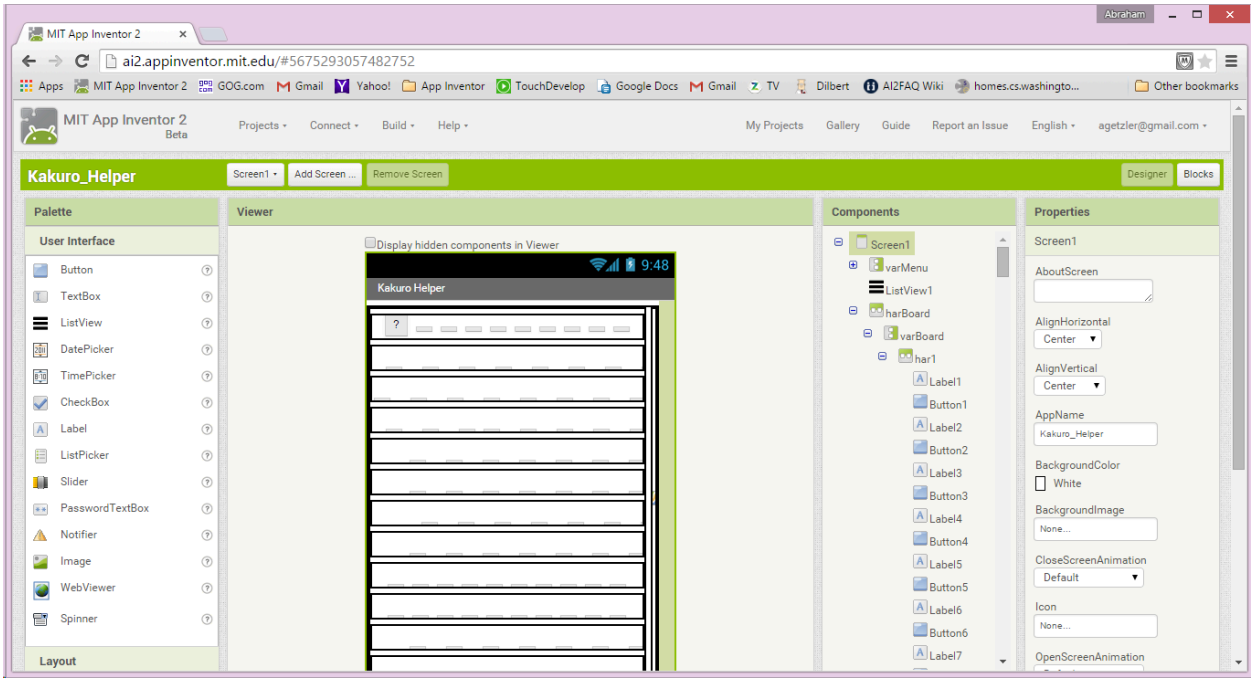
Because the white cells must contain unique numbers in the range 1 to 9, the sum of any range can't exceed 45 (1+2+...+9). If we express all range sums as two digit numbers, we can combine a vertical range sum VV and a horizontal range sum HH as the number 1VVHH, like in this Game03.csv Excel file, which was used to load the preceding example. White cells start out with a value of 0. The leading '1' in the black cells is necessary to distinguish between a black cell with two blank range values (10000) and an empty white cell (0). This lets us use numeric input for all cells.

Sample CSV Text

This is how Game03.csv looks after being exported from a spreadsheet program in csv format...

```
10000,11000,11900,10400,10000,10000,11300,11600,10000,10000
10019,0,0,0,11600,10017,0,0,13900,10500
10020,0,0,0,0,12925,0,0,0,0
10003,0,0,11110,0,0,10000,11606,0,0
10000,10900,13112,0,0,0,11717,0,0,10000
10007,0,0,0,10430,0,0,0,0,11000
10007,0,0,10817,0,0,0,10505,0,0
10000,10022,0,0,0,0,12020,0,0,0
10000,11405,0,0,10013,0,0,0,10700,11600
10016,0,0,11200,11609,0,0,11311,0,0
10024,0,0,0,0,10011,0,0,0,0
10000,10000,10017,0,0,10000,10017,0,0,0
```

The Designer



The screen is composed of a few Vertical Alignments, only one of which is visible at a time.
A Menu Vertical Arrangement is reserved for Menu operations, not yet implemented.
An off-board ListView is used for button value choices.
The board is in its own Vertical Arrangement, visible when the app is built.

The Board GUI

The board uses Labels and Buttons for its cells. Other alternatives might have worked better, like a canvas with everything drawn on it, or a grid of List Pickers. I initially tried just buttons, but I was unable to align text in buttons to the right and bottom to get the diagonal line effect.

In order to support any configuration of labels (for the black cells) and buttons (for the white cells) in 12 rows of 10 cells, I used 12 Horizontal Arrangements stacked in a Vertical Arrangement. Each Horizontal Arrangement has 10 Labels and 10 Buttons, alternating. At game load time, each pair of label and button is visited, setting one of them visible and the other one invisible depending on cell type. That will support any sequence of labels or buttons

Label Attributes

These label attributes have to be specified in the Designer, because there is no Any block support for them…

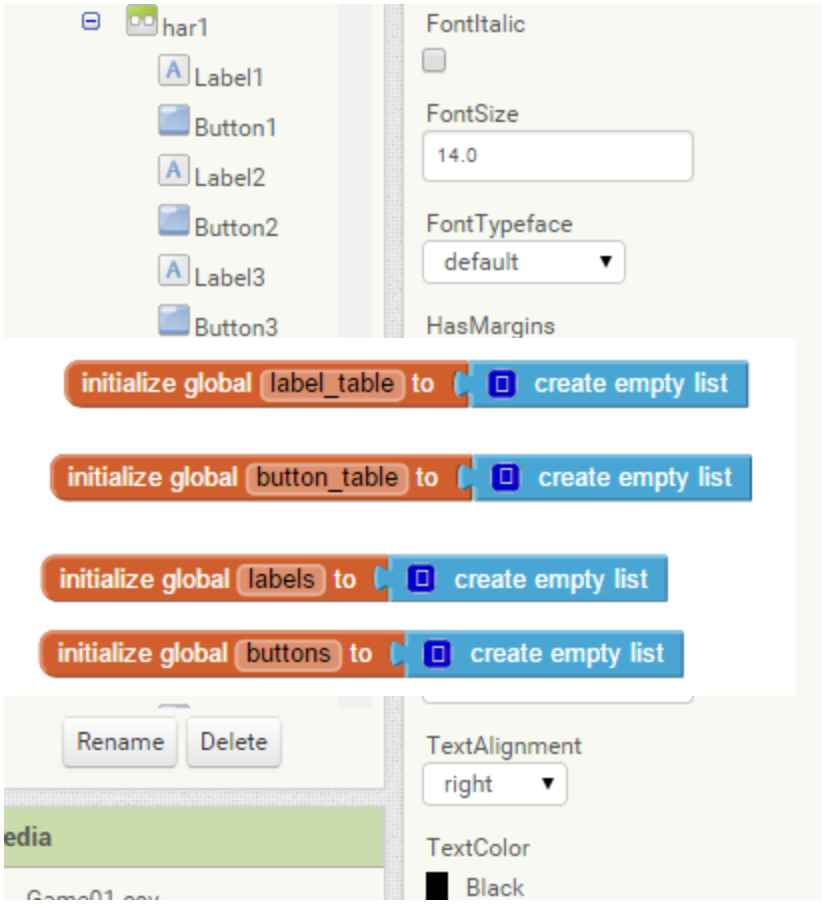
Labels must have no margins, and their Text Alignment must be Right. (Bottom would be nice, but it is unavailable.)

Label background color and text colors are set at game load time, along with height and width, which are set proportionally to screen height and width.

Button Attributes

Like labels, some button attributes must be set in the Designer because of lack of Block support…

Designer Text Alignment for buttons should be Center.



Global Variables and Data Structures

Button and Label Management Structures

For convenience in the Blocks Editor, all board buttons and labels were loaded at initialization time into the lists **buttons** and **labels**.

Once the number of rows and columns was determined (currently forced to (12,10), the buttons and labels were gathered into rows and columns in **button_table** and **label_table**.

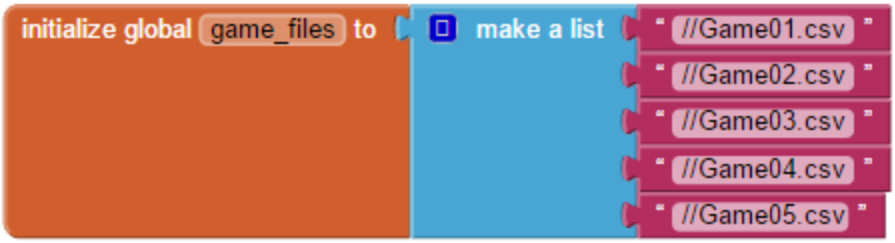
label_table

This contains the component names of the board labels, assembled into a table (list of lists, row order).

button_table

This contains the component names of the board buttons, assembled into a table (list of lists, row order).

Game Loading Structures



A small number of pre-loaded game files were uploaded to the Media drawer, to be addressed via the game_files list. This is enough to get us

started. A manifest file listing the game files would be more flexible.

Game play structures

game

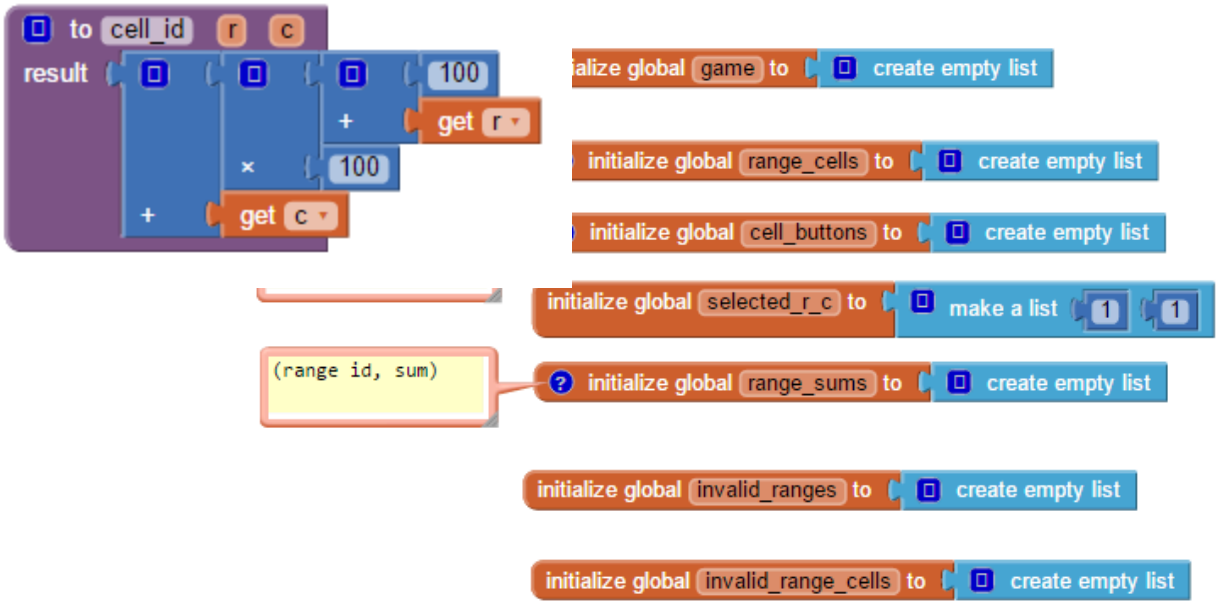
This table contains the internal representation of the board. Labels are constant integers in the form 1VVHH where VV is a two digit vertical range sum of the cell range immediately below this cell, and HH is a two digit horizontal range sum of the cell range immediately to the right of this cell. The sums can range from 00 to 45. A zero sum can arise from the range being empty because of an adjacent label cell or from being on the edge of the board. The upper left cell must always contain 10000 because row 1 and column 1 contain only labels. The player cells start out with a value of 0 and can be filled with values 1 to 9 by the player as he fills in the board.

game_history

The Undo stack will be kept here. Copies of the global **game** table are taken after every move and stacked here in a list, making it easy to undo by unstacking a level and copying that level to game. (Not yet implemented.)

Cell ID calculation

cell_id



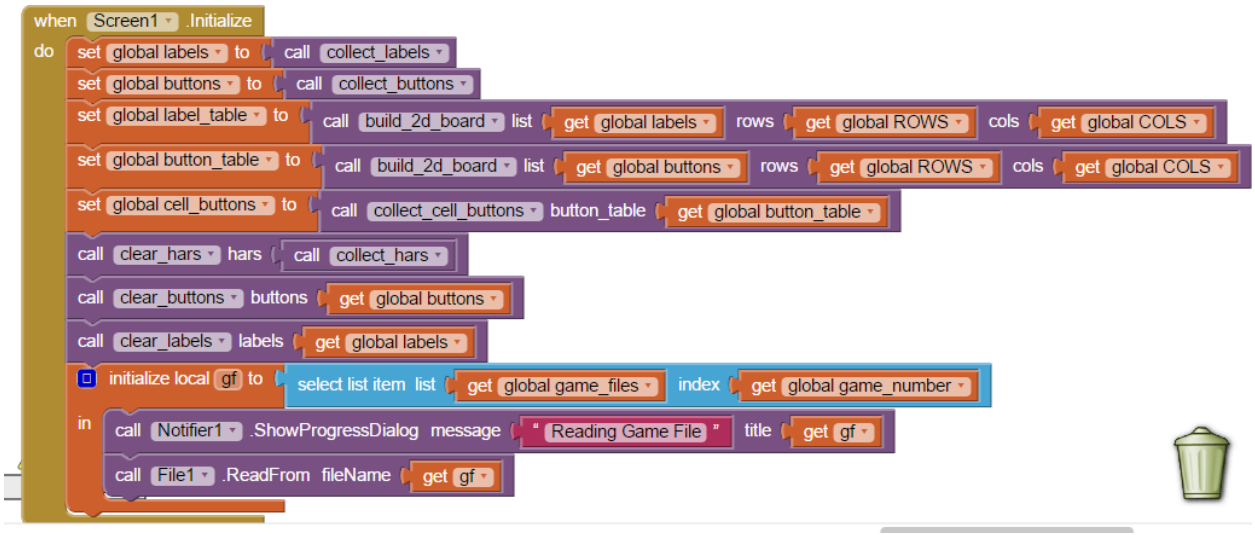
Each cell needs a way to identify it using a single value, combining both the row number RR and the column number CC of that cell. This function uses multiplication by 100 to calculate 1RRCC. The leading “1” is handy to force a double digit representation of both the row and column numbers, and to allow use of the text segment block to decode them.

Processing

Blocks Overview

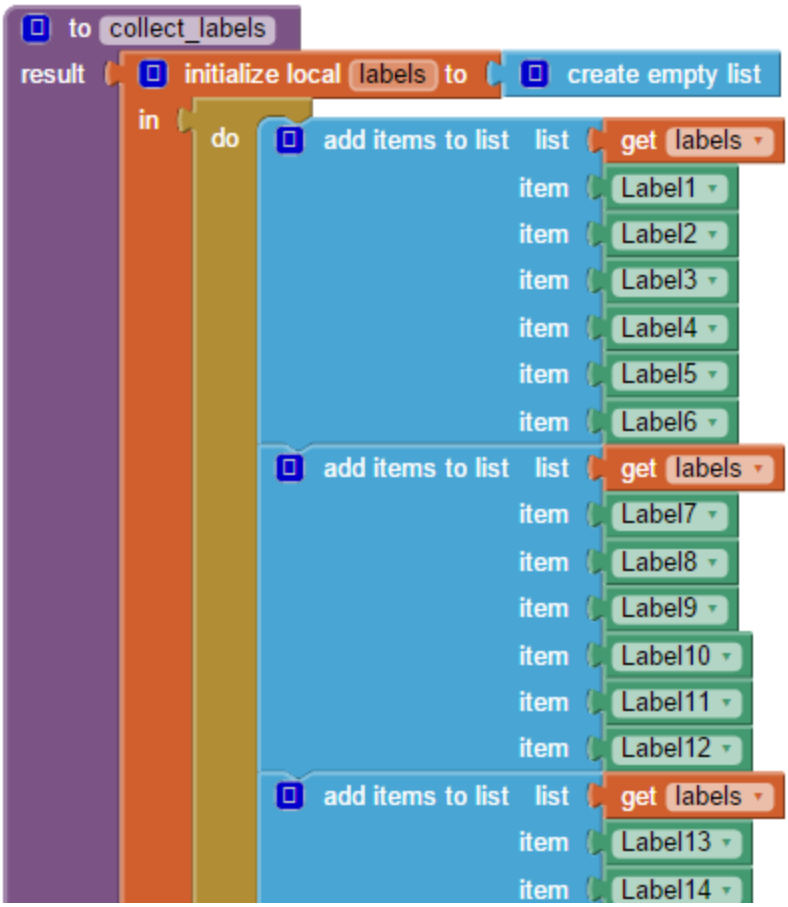
All blocks have been minimized, and separated by function.
Note: This was written before the AI2 Generic Event block functionality was added.
The grid of button Click events could collapse into a single Any Button Click event .

Screen1.Initialize



Because our labels and buttons will be addressed dynamically at run time, we need to collect them into lists at Screen1.Initialize time. The Horizontal Arrangements holding the buttons and labels are also collected in procedures A few sample game files are available in the Media drawer, and one of them is loaded via the File1 component. (Game selection to be added later.)

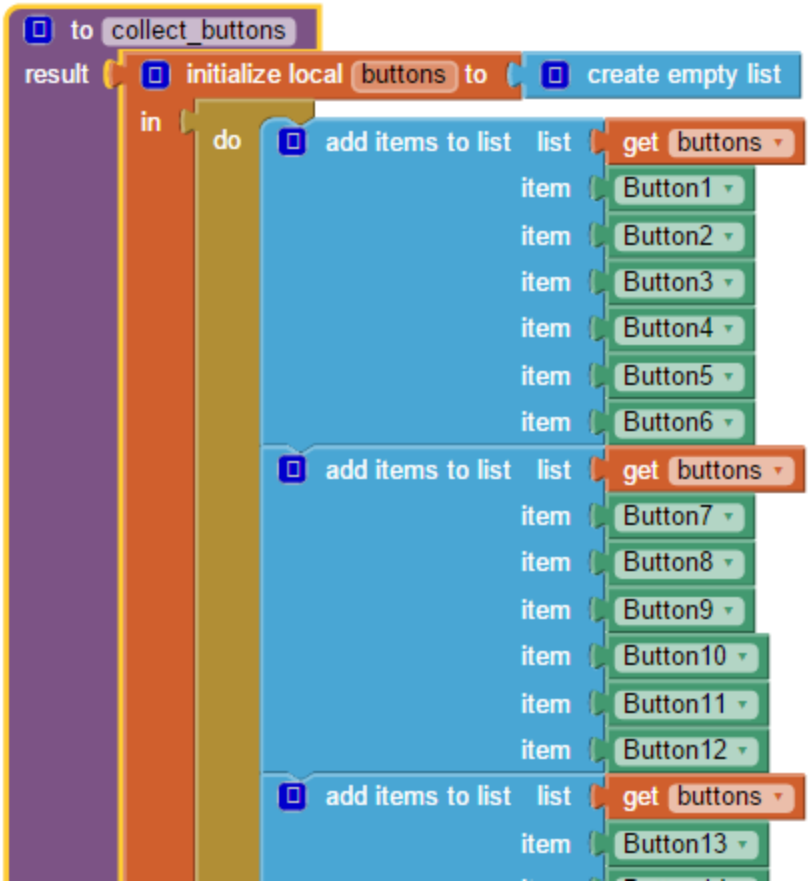
collect_labels



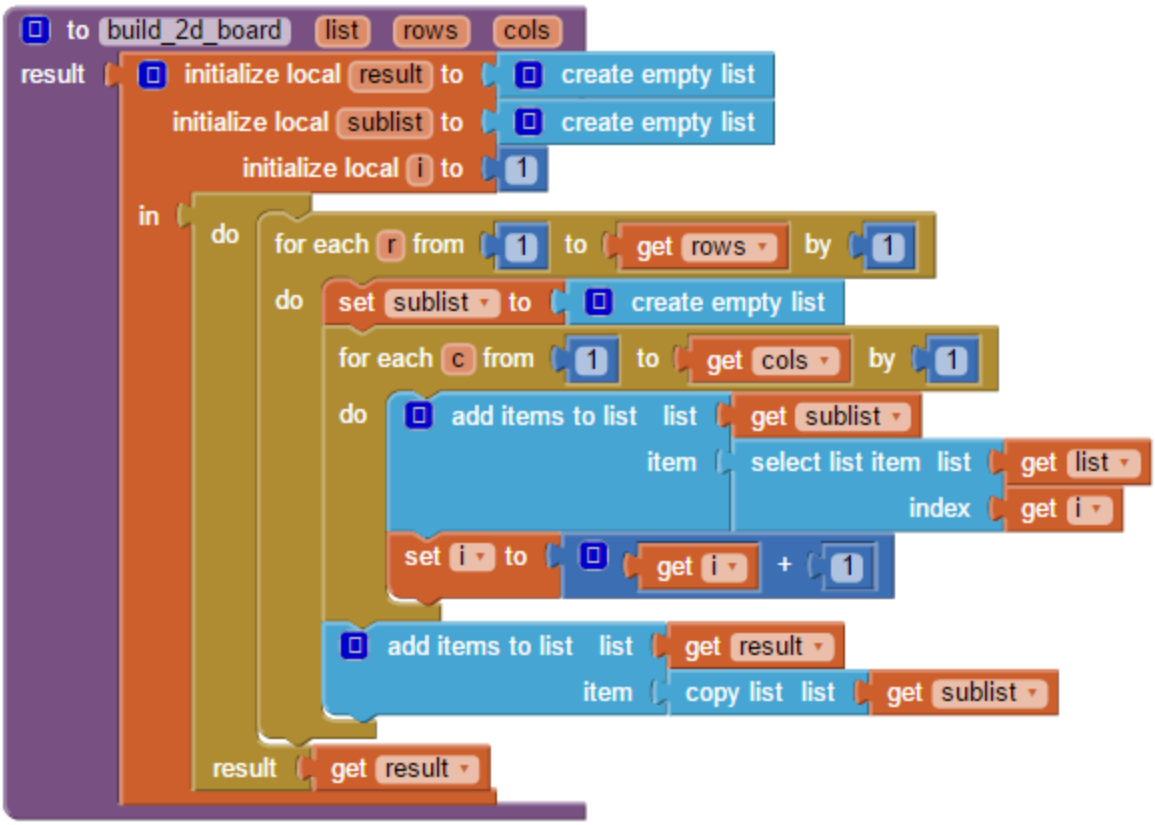
To make all label components available to the Blocks Editor, they need to be put into some kind of list structure. I took the easy way out, and collected them in row-column order into a single list, 6 at a time because my laptop can't show big blocks. I pay the price later when I re-arrange them into a list of lists to match their row & column arrangement. This is a return procedure.

collect_buttons

To make all button components available to the Blocks Editor, they need to be put into some kind of list structure. I took the easy way out, and collected them in row-column order into a single list, 6 at a time because my laptop can't show big blocks. I pay the price later when I re-arrange them into a list of lists to match their row & column arrangement. This is a return procedure.



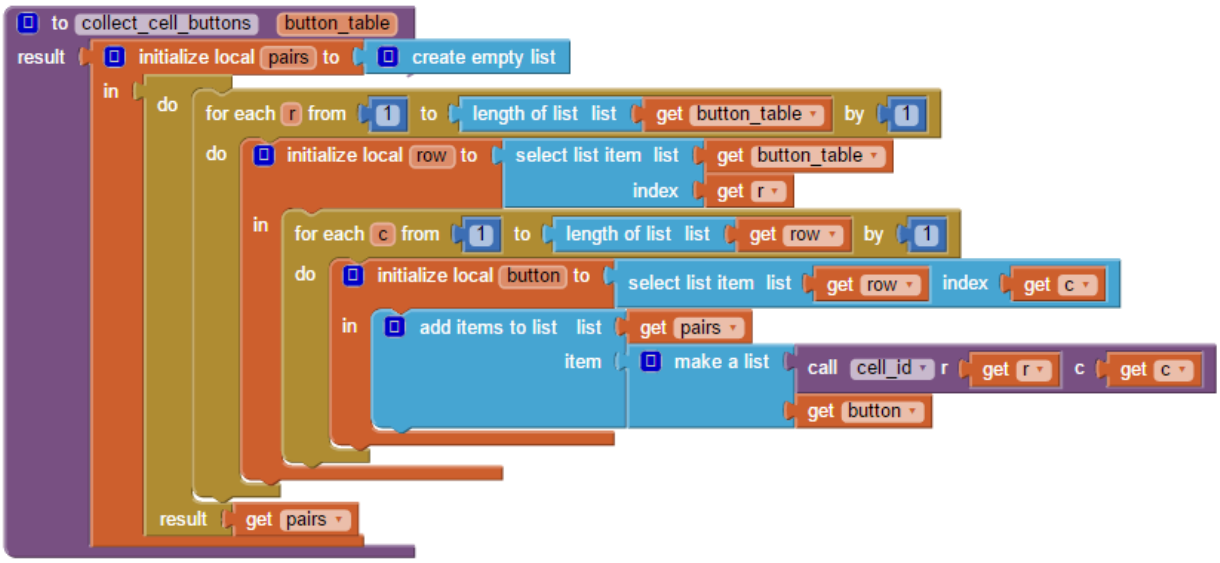
build_2d_board



For this version of the game, we have the number of rows and columns hard wired in two global variables. We reshape the one dimensional lists of labels and buttons into tables and store them into the global variables label_table and button_table .

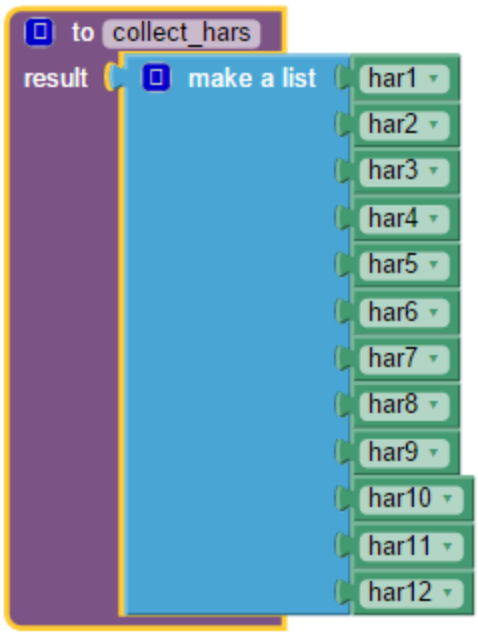
collect_cell_buttons

cell_buttons



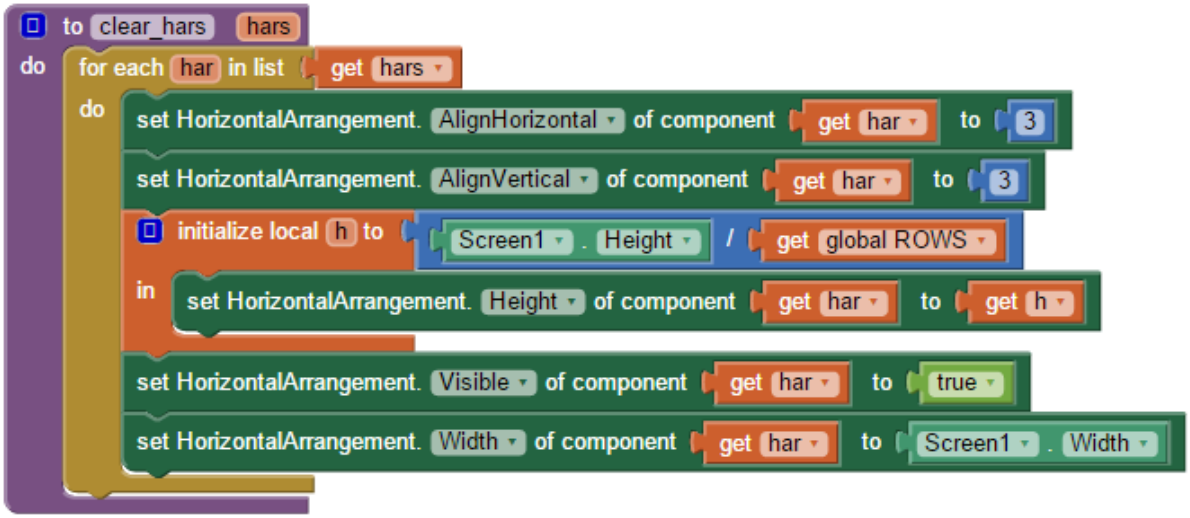
The global variable cell_buttons is initialized to a list of pairs, for use with the lookup in pairs block, to map a cell_id to its matching button component. It loops through the rows and columns of the global button_table and inserts a pair consisting of the cell_id of that row and column and that button component. This is a static structure.

collect_hars



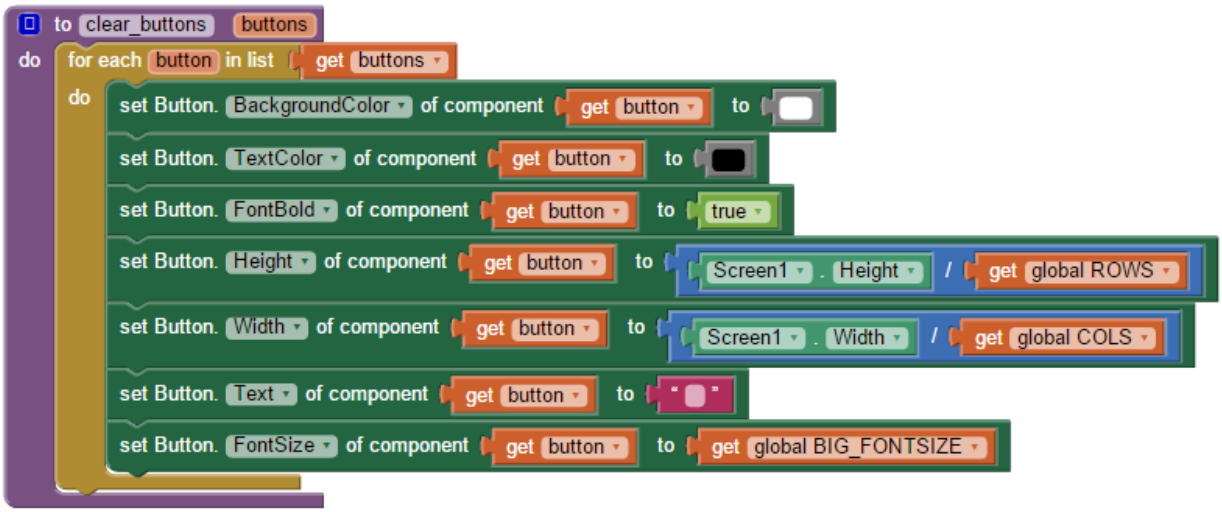
The buttons and labels are arranged in rows in horizontal arrangements, which need to be treated alike, so I collect them all into a list, and feed the list to the routine that clears them all.

clear_hars



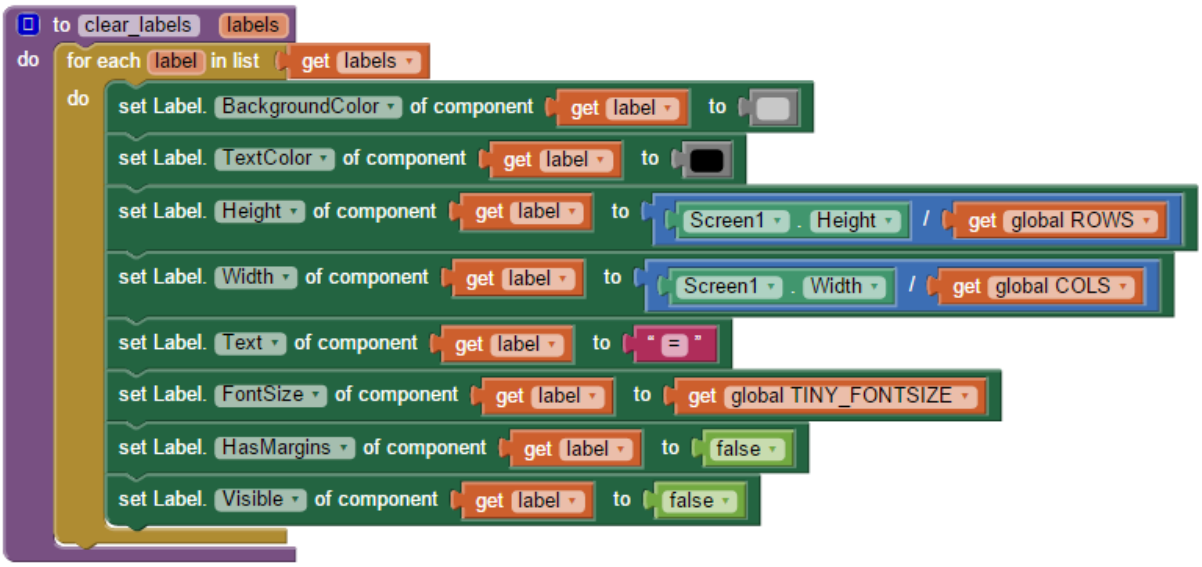
Each horizontal arrangement needs to pass down to its labels right-bottom alignment (3), and constant equal height and width.

clear_buttons



To force all the cells with buttons on the board to a common size and appearance, we run through all the buttons and set their attributes. We will customize them further after we load a puzzle.

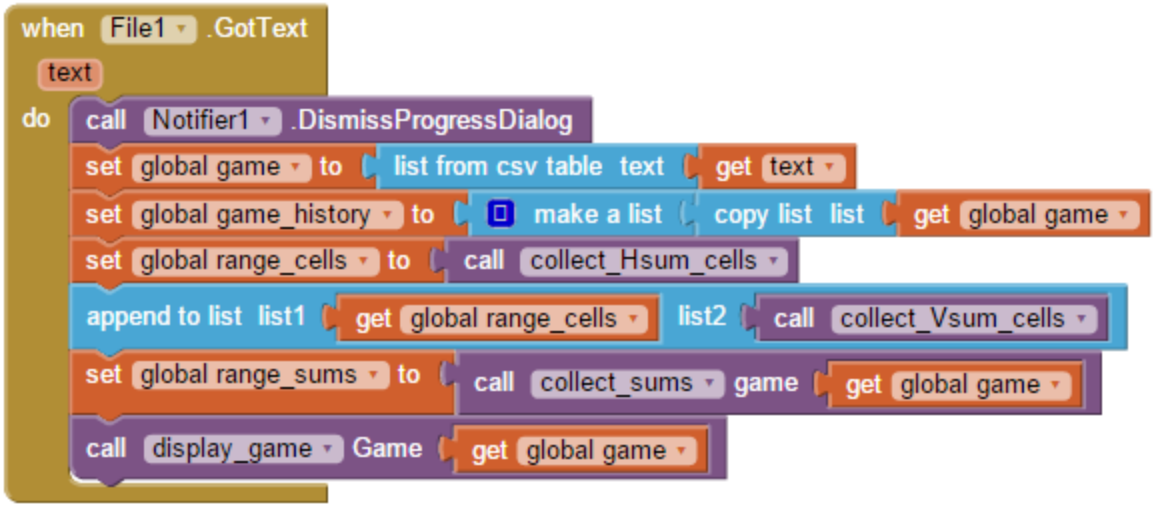
clear_labels



To force all the cells with labels on the board to a common size and appearance, we run through all the labels and set their attributes. We will customize them further after we load a puzzle.

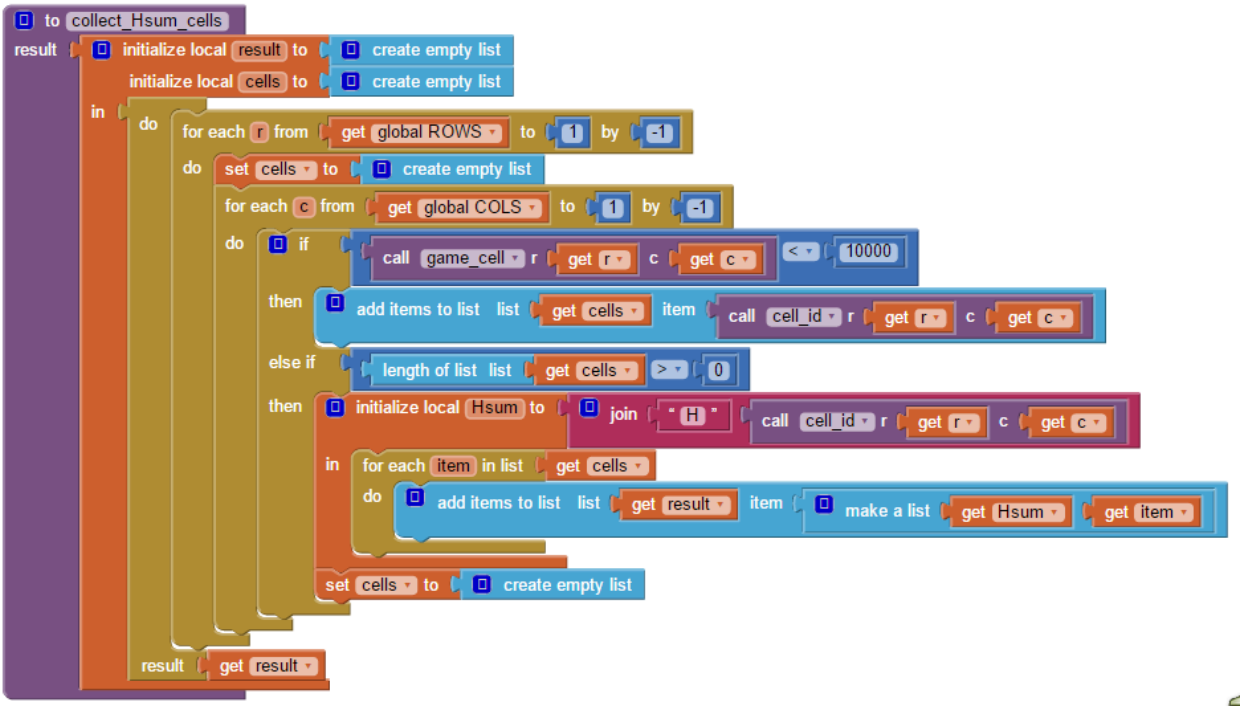
Game Loading

when File1.gotText



The text in a game file is pure numerical csv table format, ready for a **list from csv table** block. We initialize game and game_history , collect range cells horizontally and vertically, calculate range sums, and display the game board.

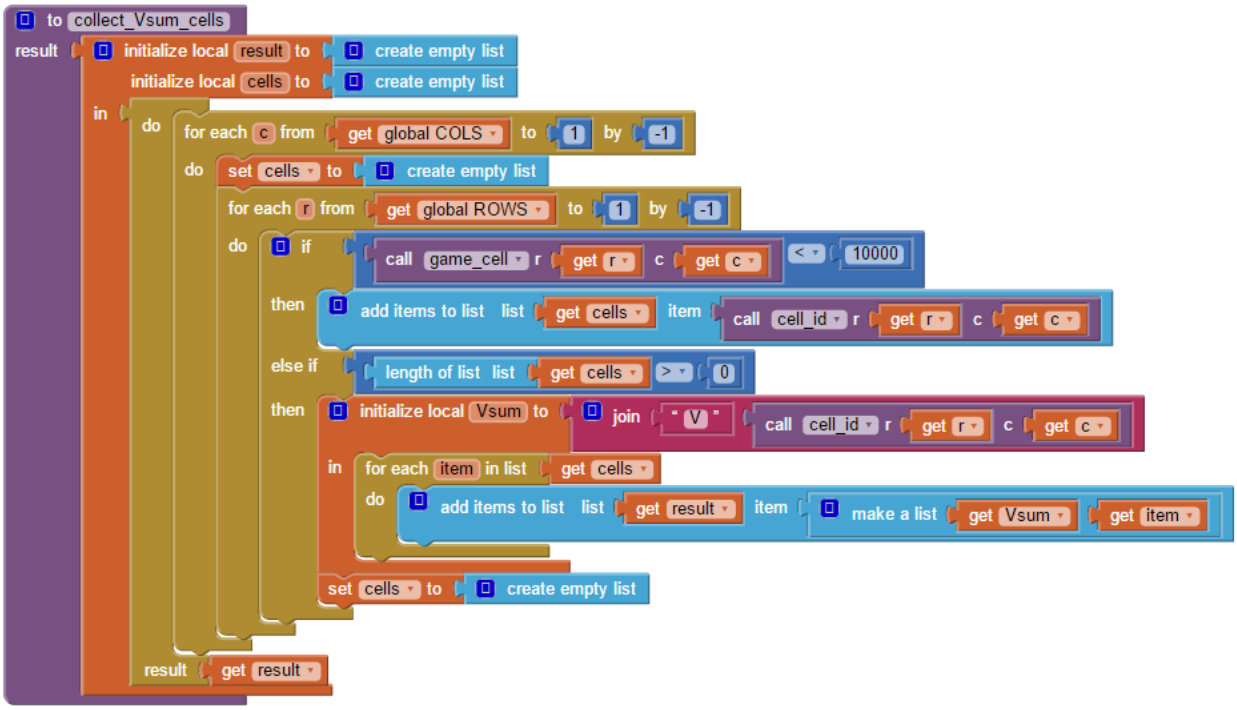
collect_Hsum_cells



The result of this procedure will be a list of pairs (HRRCC, cell_id) where HRRCC identifies the row RR and column CC of the sum of the horizontal range that includes the cell identified by cell_id .Ranges usually have several cells in them, so there will be several pairs with the same HRRCC value but different consecutive cell_id values.

The ranges are identified by scanning game from right to left, building up an empty list of cells until a sum cell is hit, then posting the pairs for that range, clearing the list, and continuing up the row for the next range. Empty ranges are skipped.

collect_Vsum_cells



The result of this procedure will be a list of pairs (VRRCC, cell_id) where VRRCC identifies the row RR and column CC of the sum of the vertical range that includes the cell identified by cell_id .Ranges usually have several cells in them, so there will be several pairs with the same VRRCC value but different consecutive cell_id values.

The ranges are identified by scanning game from bottom to top, building up an empty list of cells until a sum cell is hit, then posting the pairs for that range, clearing the list, and continuing up the column for the next range. Empty ranges are skipped.

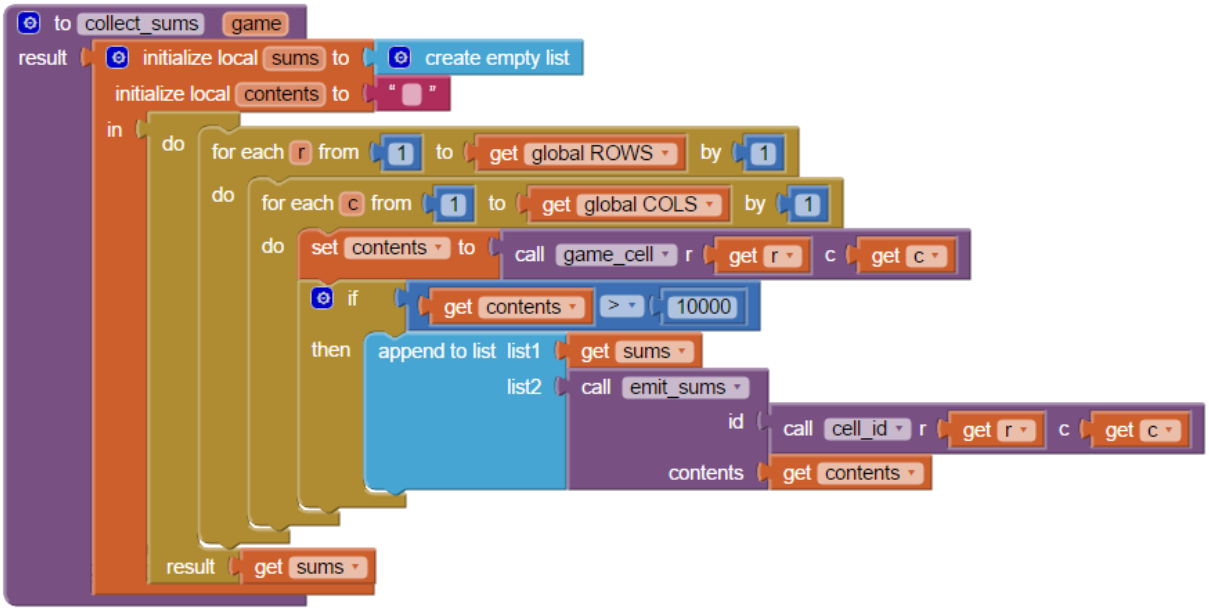
Global range_cells

The horizontal and vertical range-cell pairs are collected into a single global list of pairs, range_cells.

Global range_sums

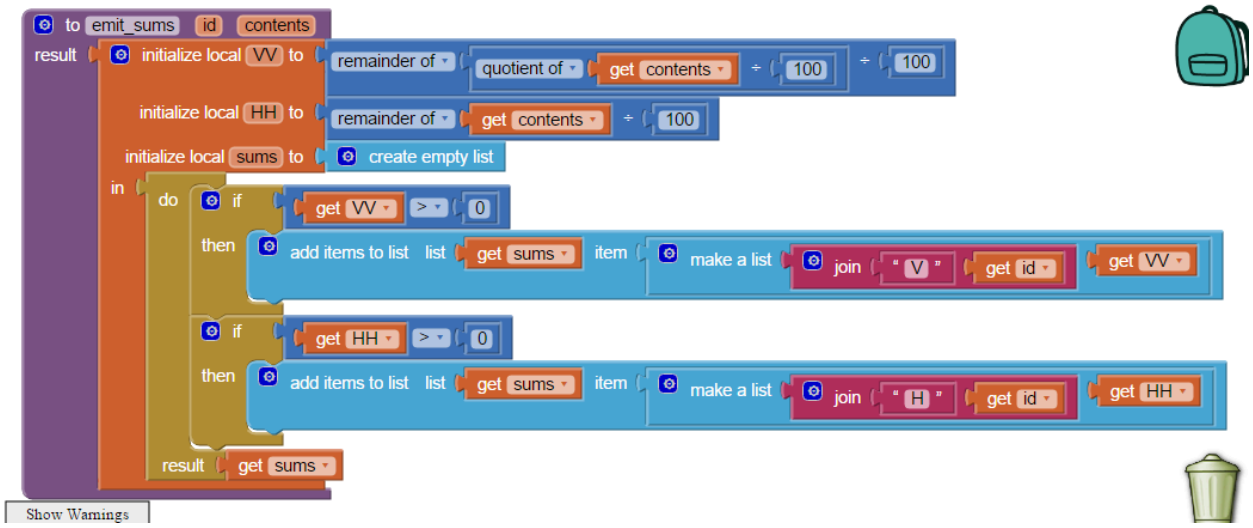
This global list is a 2 column table mapping range names into the target sums.
We build it at game load time from procedure collect_sums.

Collect_sums



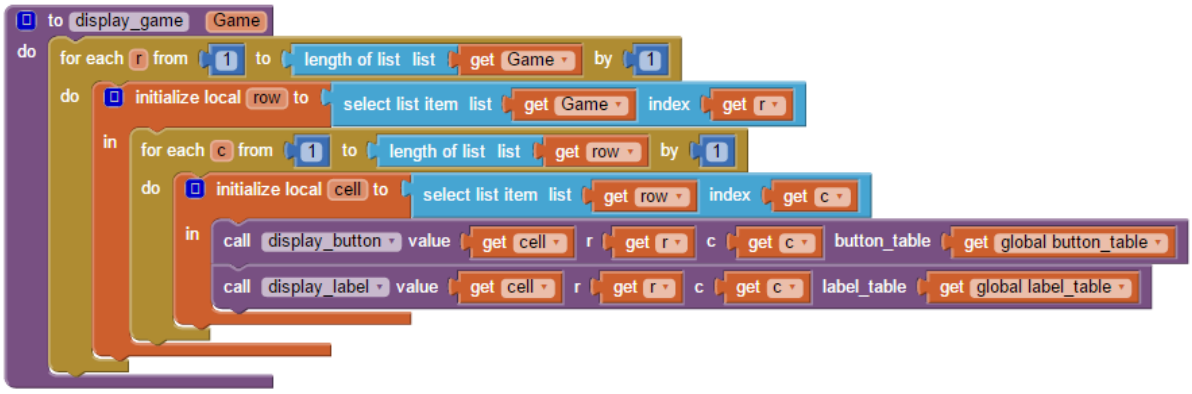
This procedure traverses a given game table, by row and column, checking each game_cell if its **contents** are of the form 1HHVV, adding to a return list **sums** the extracted values from procedure **emit_sums .**

Emit_sums



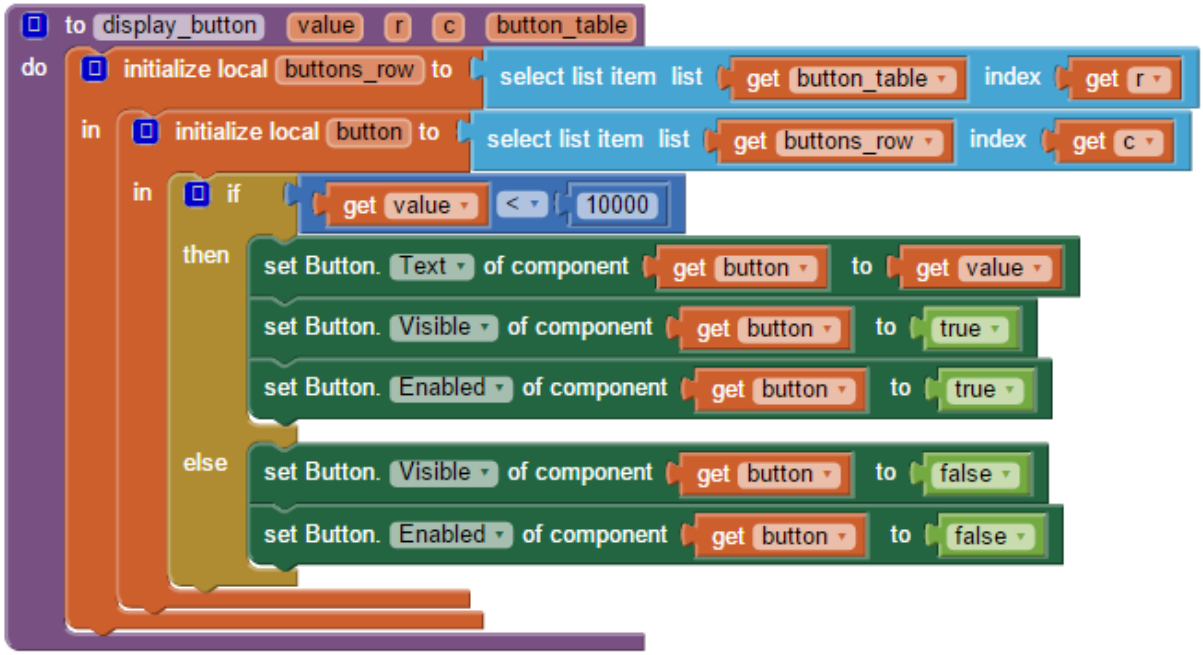
These range sums are going to be needed for later, when the user works the puzzle.
The VV (vertical) and HH (horizontal) sum parts of the given range cell are extracted, and are identified by a range type letter V or H prefix to the range cell ID, paired with the sum value.
The (id,sum) values are collected in pairs, suitable for use with the lookup in pairs block.

display_game



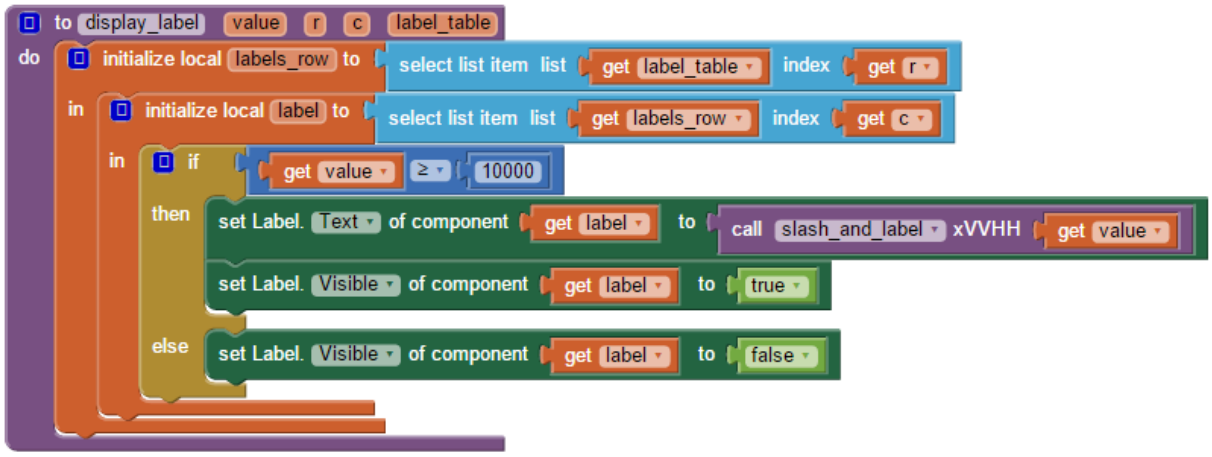
To display the game table, we scan each row r and column c, and pass that cell, button, and label to routines display_button and display_label, which will decide to display or hide their respective component based on the cell value.

display_button



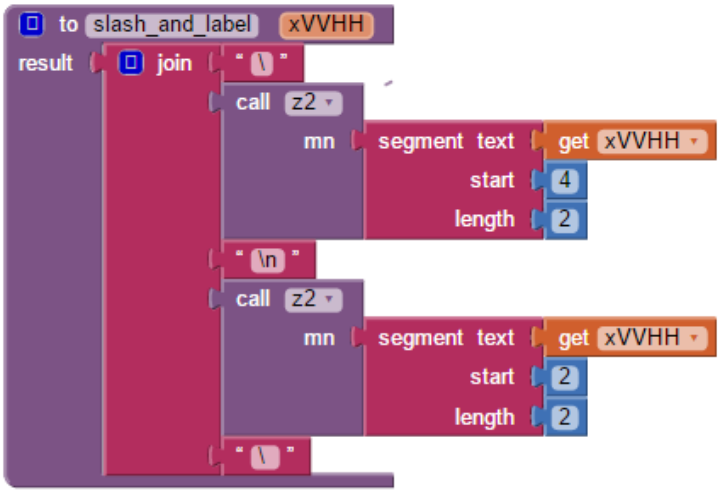
To display a button, we start by looking up its component in the button_table , by row and column, verifying that it is indeed a button by value < 10000, then set the button contents, visibility, and enabled attributes accordingly.

display_label



To display a label, we start by looking up its component in the label_table , by row and column, verifying that it is indeed a label by value >= 10000, then set the label contents, visibility, and enabled attributes accordingly. The diagonal slash separating the vertical and horizontal sums is built up in the slash_and_label value procedure, and requires a monospace font and right-bottom alignment.

slash_and_label

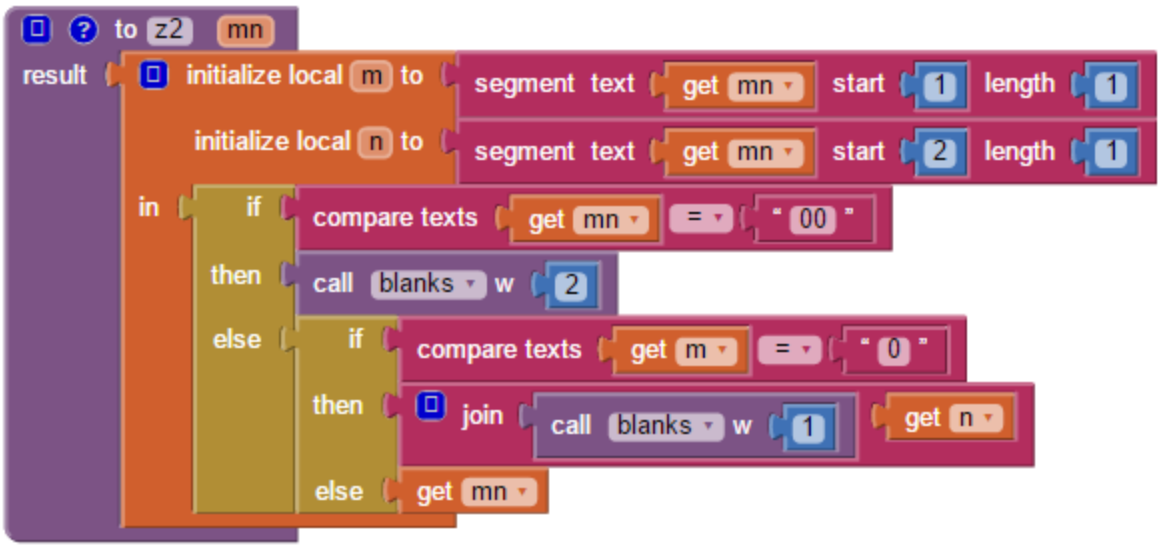


This routine will take a 5 digit number 1VVHH and show it as

```
\VV
HH\
```

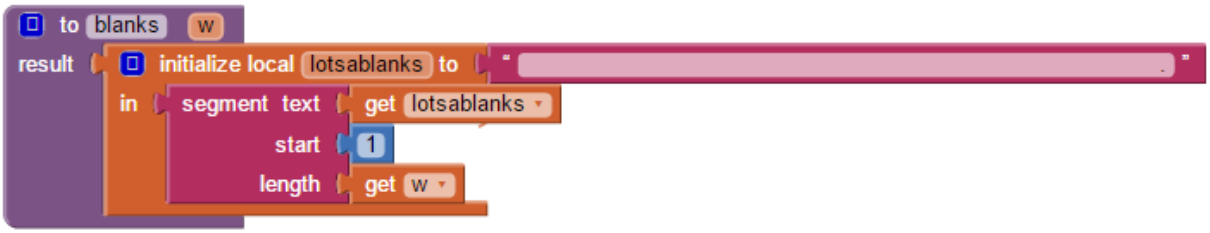
The routine Z2 handles forcing a constant width of 2 places, with leading blanks.

Z2



This routine replaces all leading blanks of a two digit number, replacing them with a blank. Because the Block Editor seems to eat blanks from text blocks, I had to write a value function that would return a requested number of blanks …

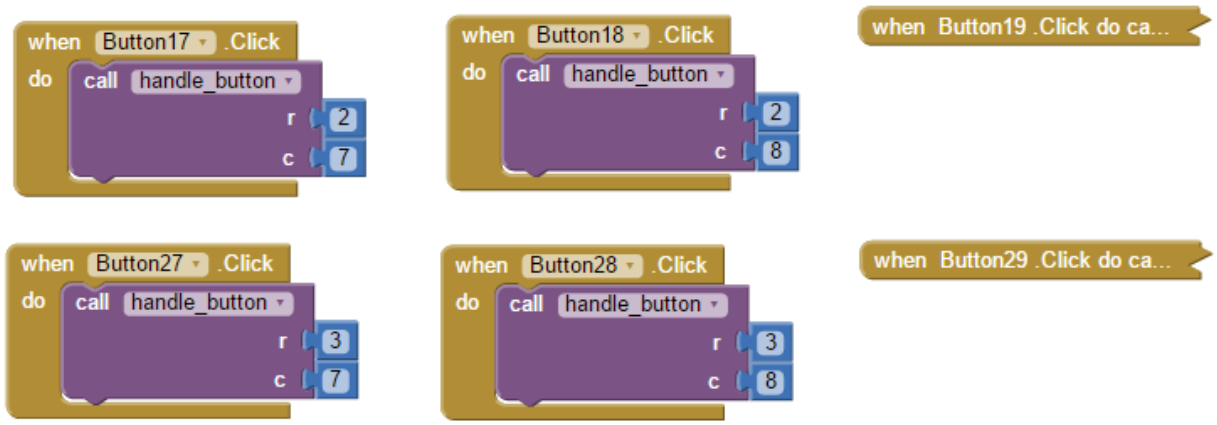
blanks



Game Play

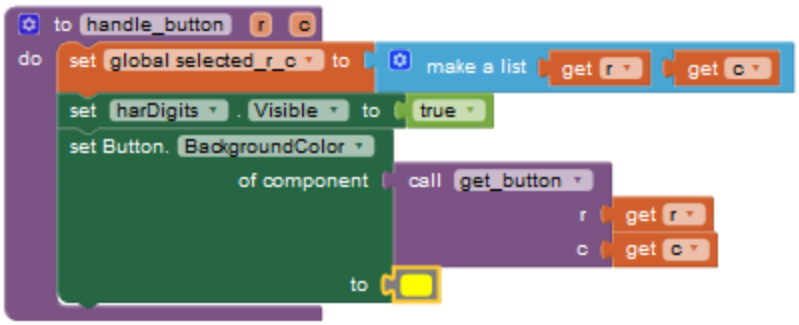
To play the game, the user pushes one of the buttons on the board, then selects a ListView selection for what digit he wants to fill in for that cell, (or some menu options, to be added later.)

when_Button_rc_Clicked



handle_button

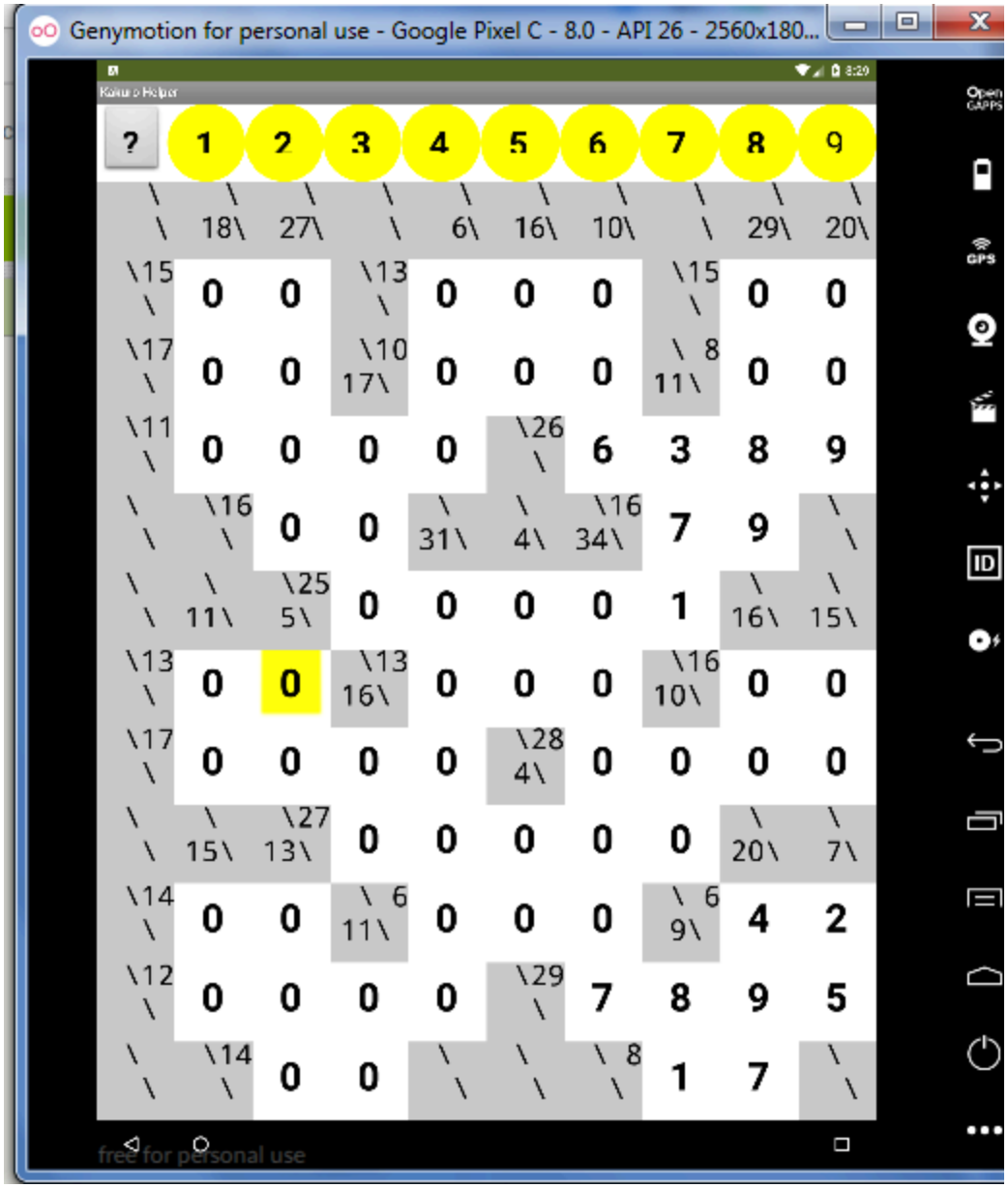
global selected_r_c



Each Button Click handler calls handle_button , which saves the selected row and column number in global selected_r_c as a pair, for later use after ListView1 returns a Selection. The button at (r,c) is looked up by routine get_button, then is colored yellow, so the player can choose a digit for that cell.

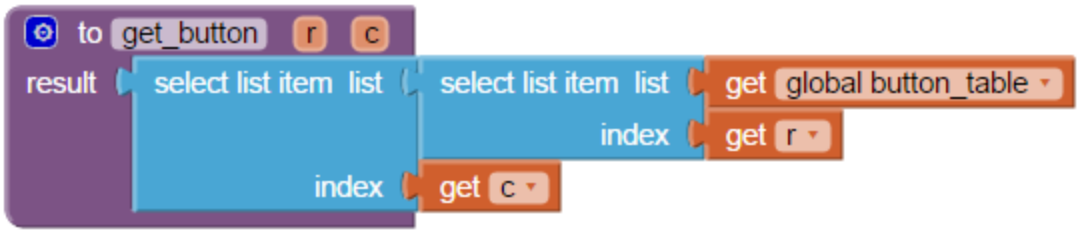
There are lots of alternative ways to have handled this, like a table of active and inactive text boxes.

The row and column of the selected cell are highlighted in yellow:



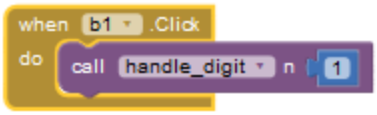
Notice how the input digits are exposed at the top, and also colored yellow.

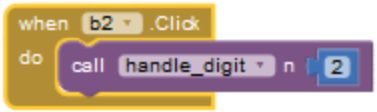
Get_button



This is a 2 dimensional lookup into table button_table, initialized earlier.

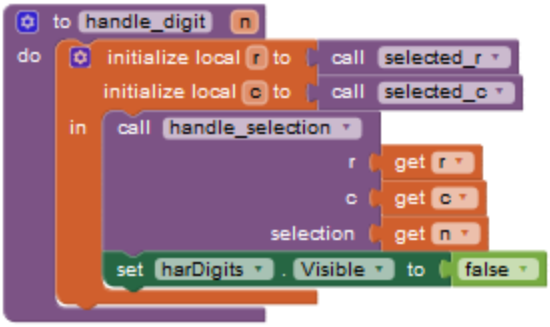
When b1...b9 Clicked





Digit entries are handled by the handle_digit procedure, taking as input the requested value to be assigned at the highlighted cell as recorded in the global selected_r_c variable.
Because I needed to fit a Menu button onto the board at the left of digits, 1-9, clearing a cell to 0 is handled under the Menu button by the btnClear button.

Handle_digit

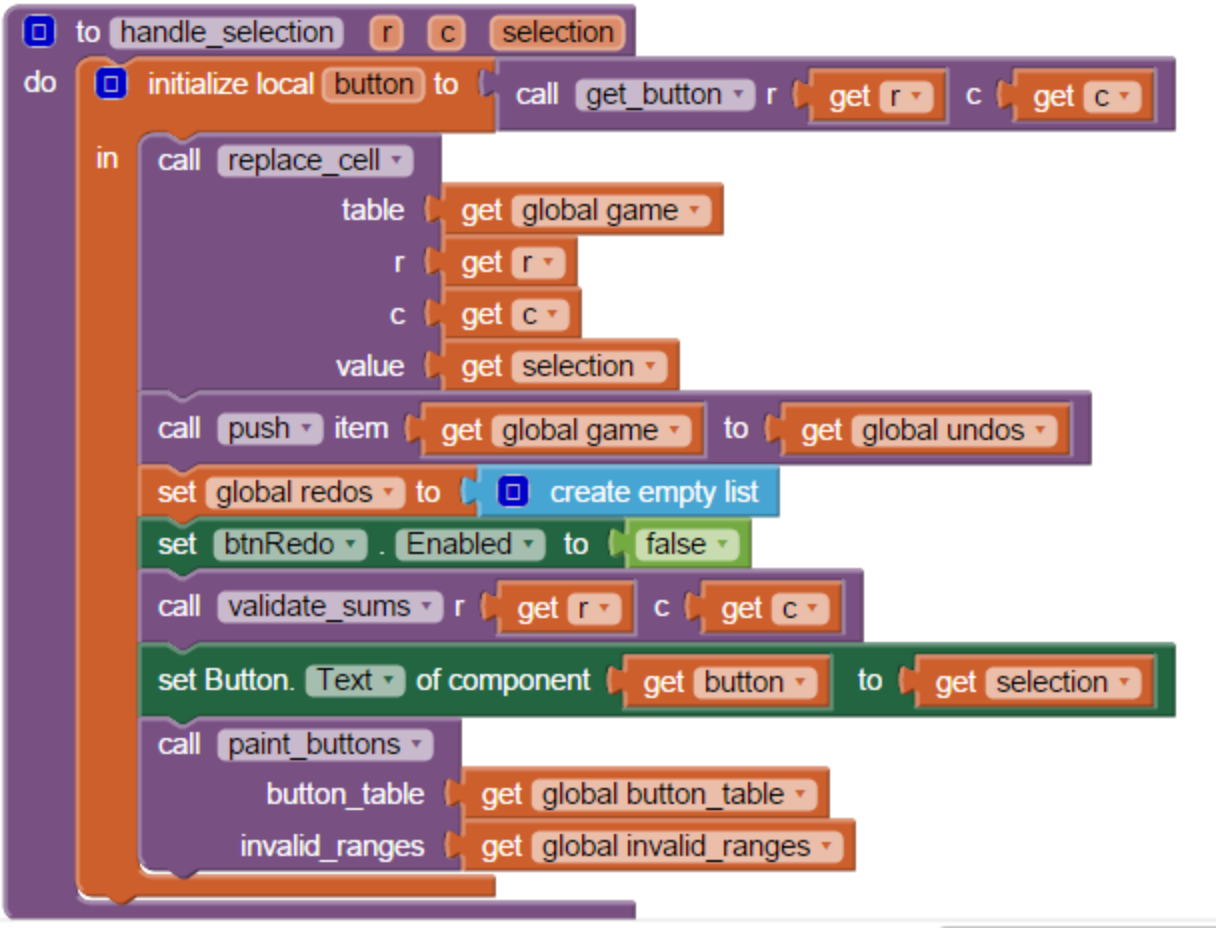


The first level of functionality for setting a cell value, identifying the row and column, is done by procedures selected_r and selected_c. Then we call handle_selection with the row, column, and desired value that was just selected. Afterwards, we hide the Horizontal Arrangement with the input digits.

Selected_r

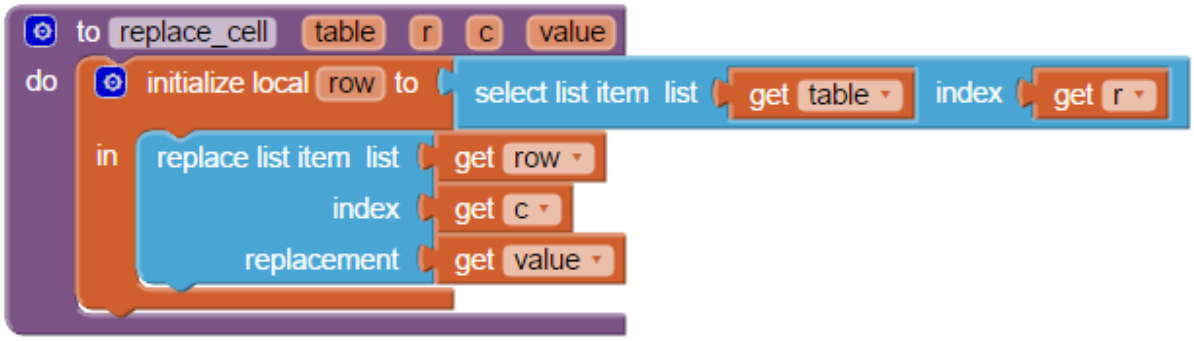
Selected_c

Handle_selection



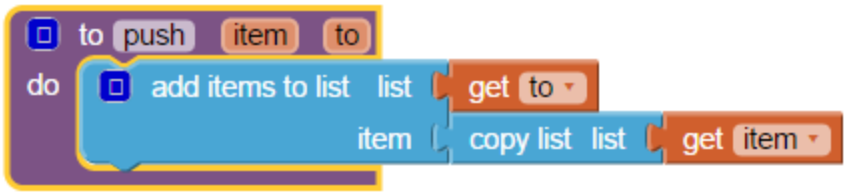
To handle selection of a value for a cell, we first have to identify which button is associated with that (r,c) coordinates, using procedure **get_button**, and save it in local variable **button**.
We then call procedure replace_cell to update our **game** board, then we **push** the new copy of the game board onto our global **undos** stack. A new move clears the global **redos** stack .
We disable the Redo button (hidden in the Menu) until we have something to redo.
We call procedure **validate_sums** to collect the ranges that need to be highlighted because they are clearly in error. The selected **button** has its .**Text** set to the requested **selection** value.
We then call **paint_buttons** to refresh each button with its validity indicator, white (valid) or red (invalid) background.

Replace_cell



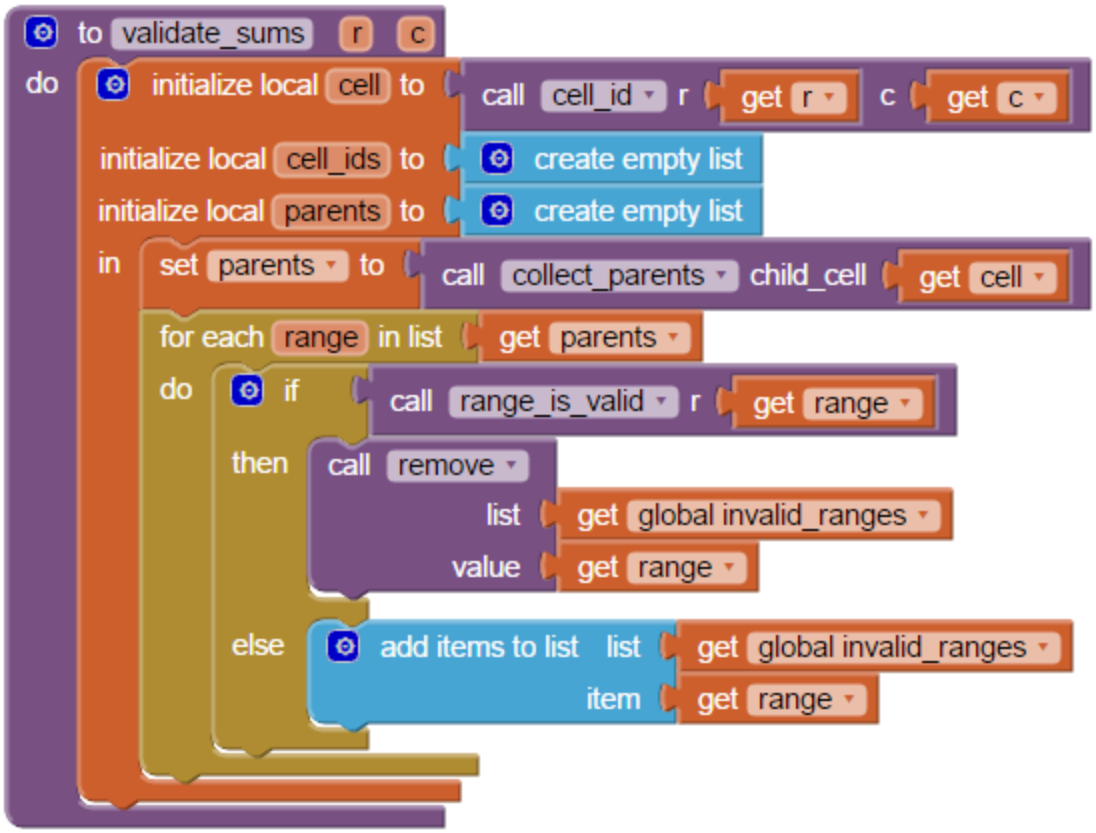
Cell value replacement is done in place in a two dimensional table (list of lists).

Push



Pushing and popping are set up to use the last (highest index value) item of the stack as the most recent.

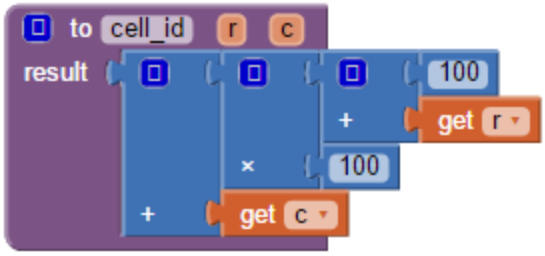
Validate_sums



Validation of ranges is done incrementally, just for the two ranges (horizontal and vertical) that contain the last updated cell. A global list of invalid ranges is maintained for the game board, for later use painting invalid cell ranges. This is a bit shaky, done for speed, to avoid duplicate work. If that global invalid_ranges list gets out of sync with the board, error painting will be thrown off.

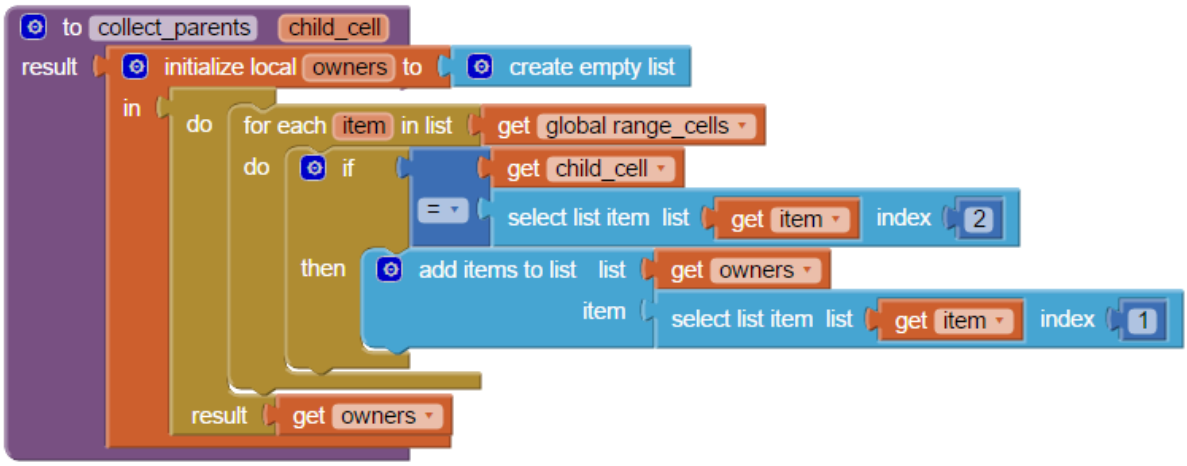
We identify the **cell** using a single value, returned from procedure cell_id from its coordinates **r**,**c**. We call collect_parents to gather the parent ranges of our **cell** into local list **parents**. For each **range** in the list of **parents** we test if **range_is_valid.** If valid, we **remove** it from the global list invalid_ranges, otherwise we add it to that list.

Cell_id



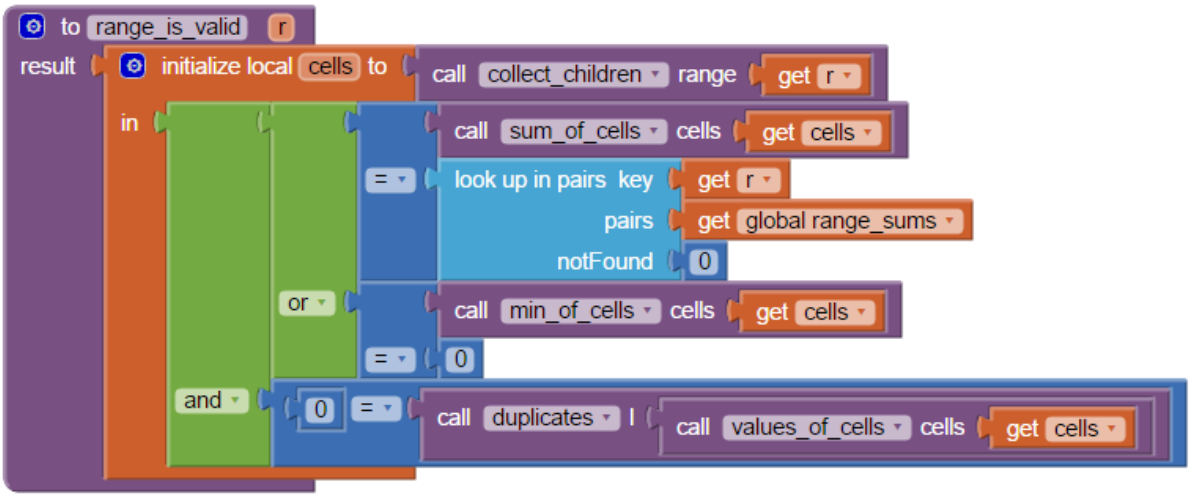
The id of a cell is a number of the form 1RRCC where RR is the row r, and CC is the column **c**, as 2 digit numbers. This gives us a single value, for easy comparison.

Collect_parents



This procedure references global range_cells, a two column table (range,cell) which was built during game loading. It builds up an initially empty list of **owners** from column 1 if column 2 matches the given cell id.

Range_is_valid

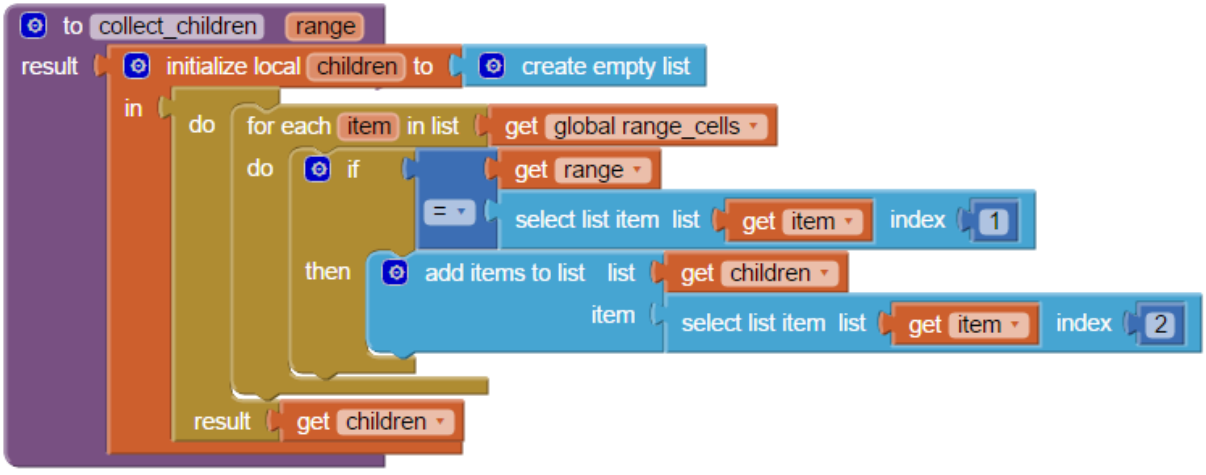


For the game Kakuro, there are 2 rules that are applied to ranges that have been filled in (no zeroes):

1. No duplicates allowed
2. The sum must match the stated target value for that range.

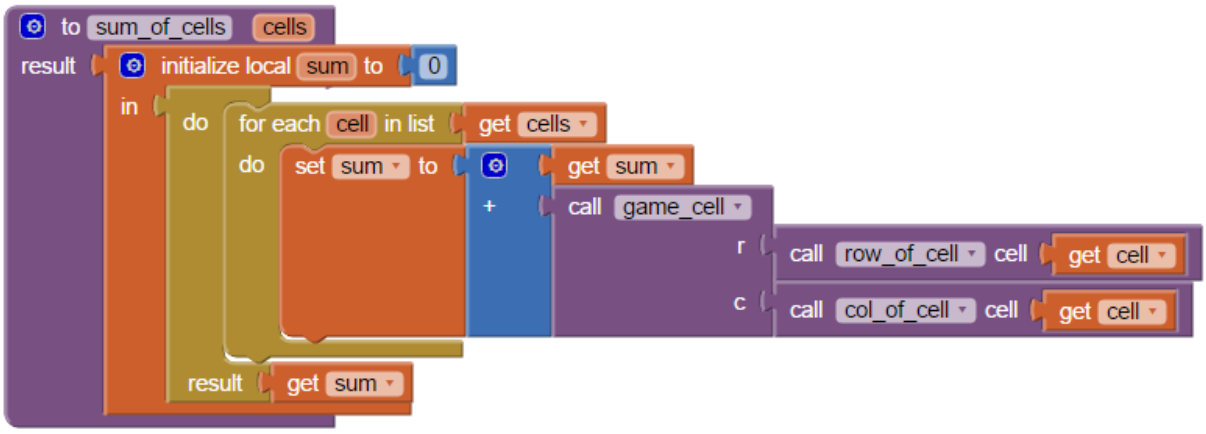
First we collect the children of the given range r into local variable **cells**, a list of their cell ids. We count the number of duplicates of the values of those cells. There should be 0 duplicates. Then we check if we have a full range (**min_of_cells** > 0) and if the **sum_of_cells** of that range matches the value for that range in global range_sums.

Collect_children



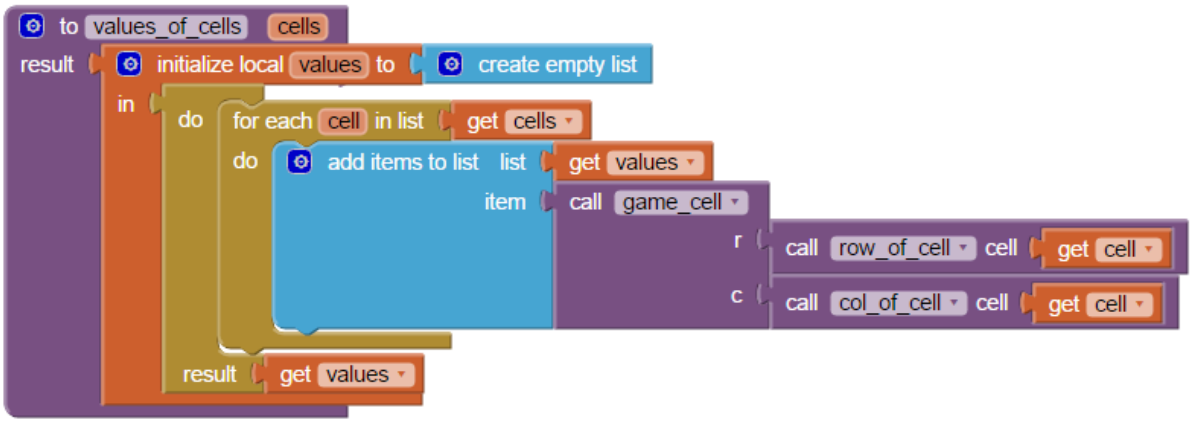
This procedure traverses the global range_cells, returning a list of all the cells that are in the given range.

Sum_of_cells



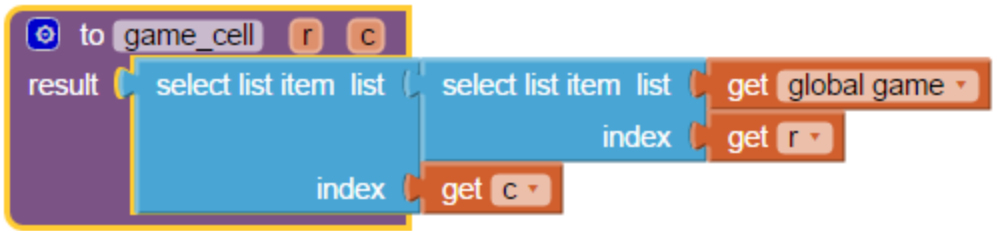
This procedure does a running subtotal of the values returned from game_cell at the row_of_cell and col_of_cell for the given **cells** list.

values _of_cells

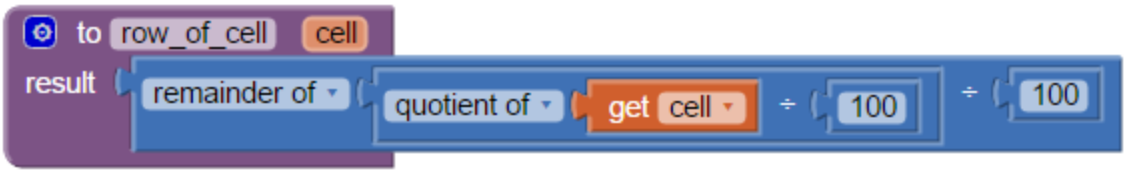


To collect the **values** of a list of **cells**, we look up each game_cell value for the row_of_cell and col_of_cell of that cell and add it to our return list of **values**.

Game_cell

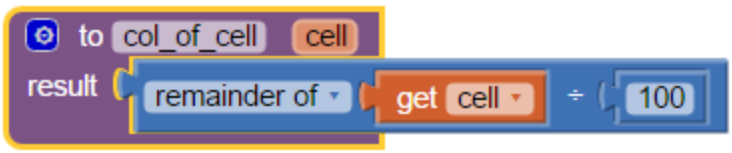


Row_of_cell



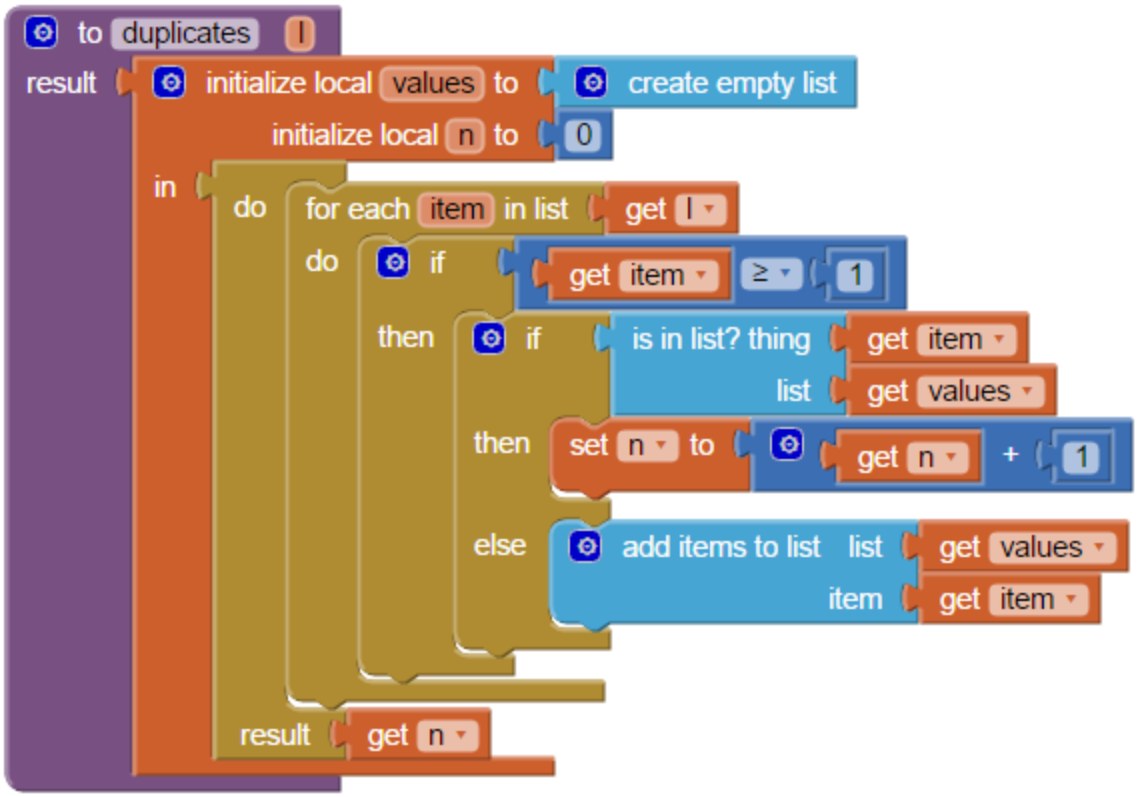
Cell IDs are of the form 1RRCC, so this extracts the RR.

Col_of_cell



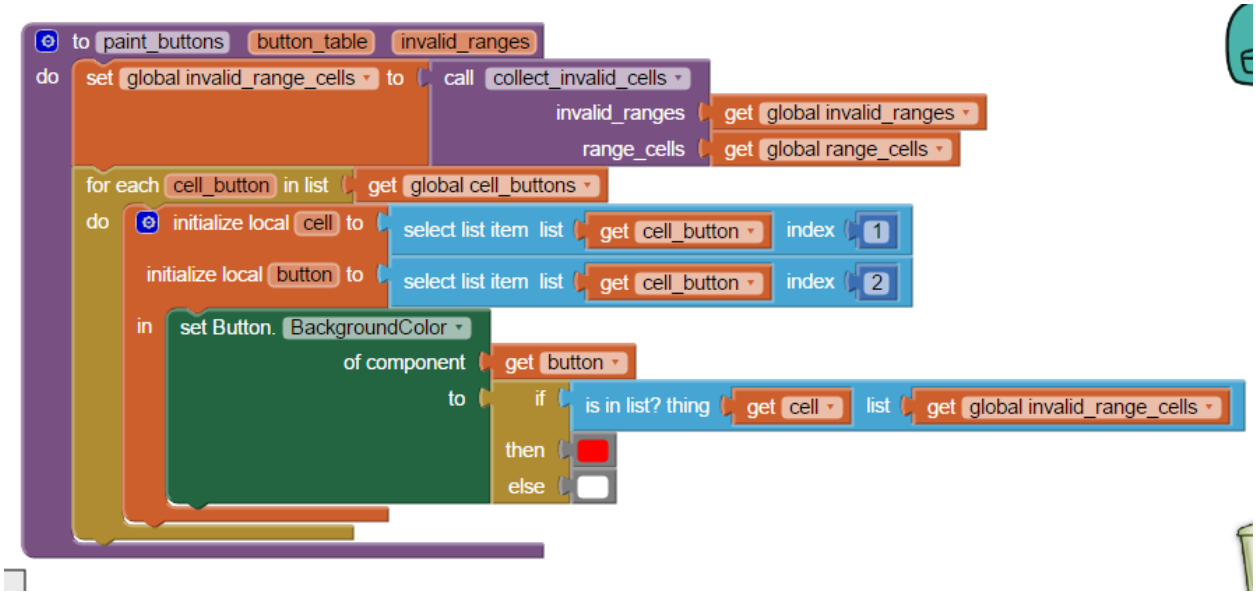
Cell IDs are of the form 1RRCC, so this extracts the CC.

Duplicates



To calculate the number of non-zero duplicates in a list, we build up a temporary list of values we have seen already as we traverse the input list, adding 1 to a return count **n** each time we get a hit.

Paint_buttons

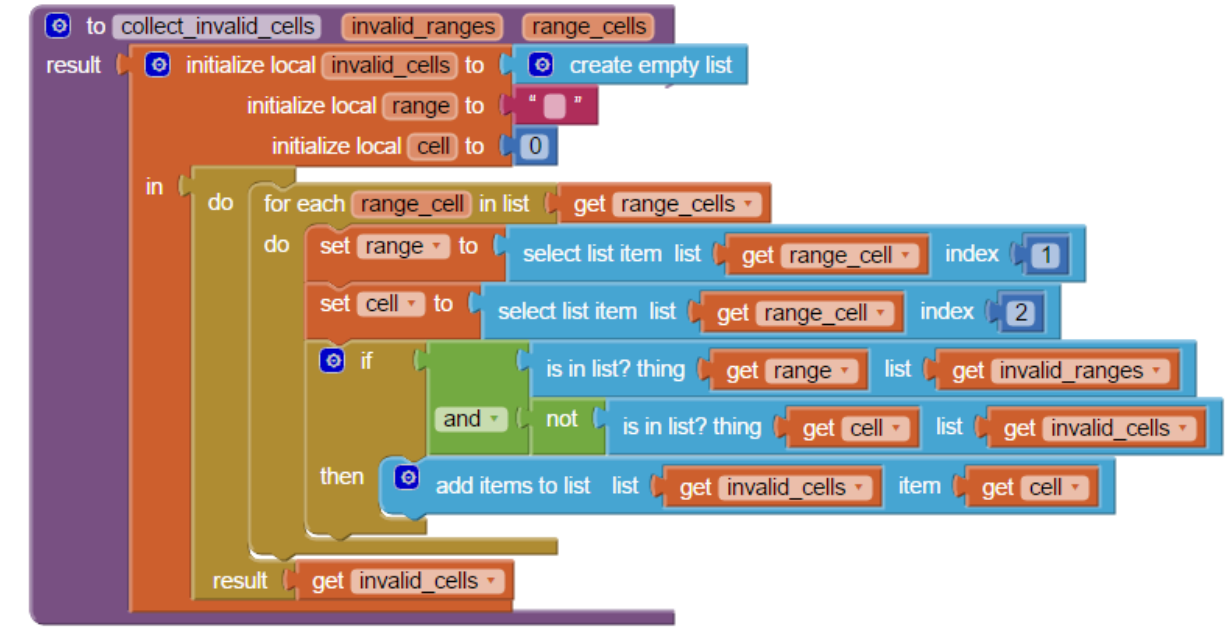


This procedure first builds a global list invalid_range_cells using procedure collect_invalid_cells based on the global lists invalid_ranges and range_cells. It then examines each **cell_button** in the global list cell_buttons, breaking out its **cell id** and its **button** component and setting the button's BackgroundColor to red if the cell id is in the global invalid_range_cells, or white otherwise. (This has the desirable side effect of clearing the yellow highlight of a cell button that was just changed.)

Global invalid_ranges

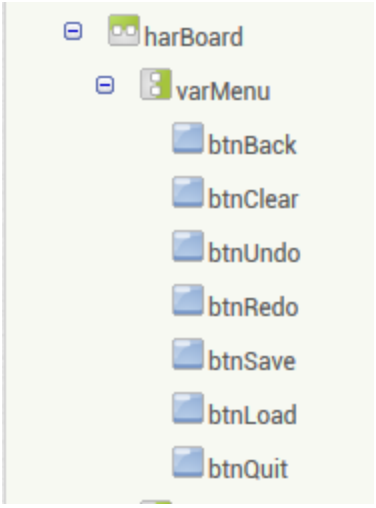
Global invalid_range_cells

Collect_invalid_cells



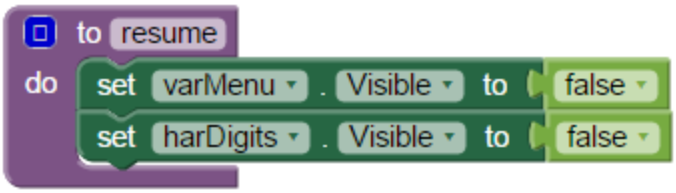
Invalid cells are the cells whose cell ids are in invalid ranges. We traverse the given list of range_cell pairs, extracting the range and contained cell id. If the range is not already in the list of invalid ranges and if the cell is in that range, we add it to the returned list.

Menu Buttons



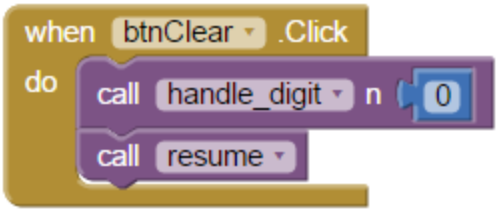
When btnBack Clicked

Resume



The Back button just calls the resume procedure, which hides the Menu and Digits Arrangements.

When btnClear Clicked



The Clear button hides the Menu and acts as if a 0 digit had been entered for a cell value, using procedure handle_digit.

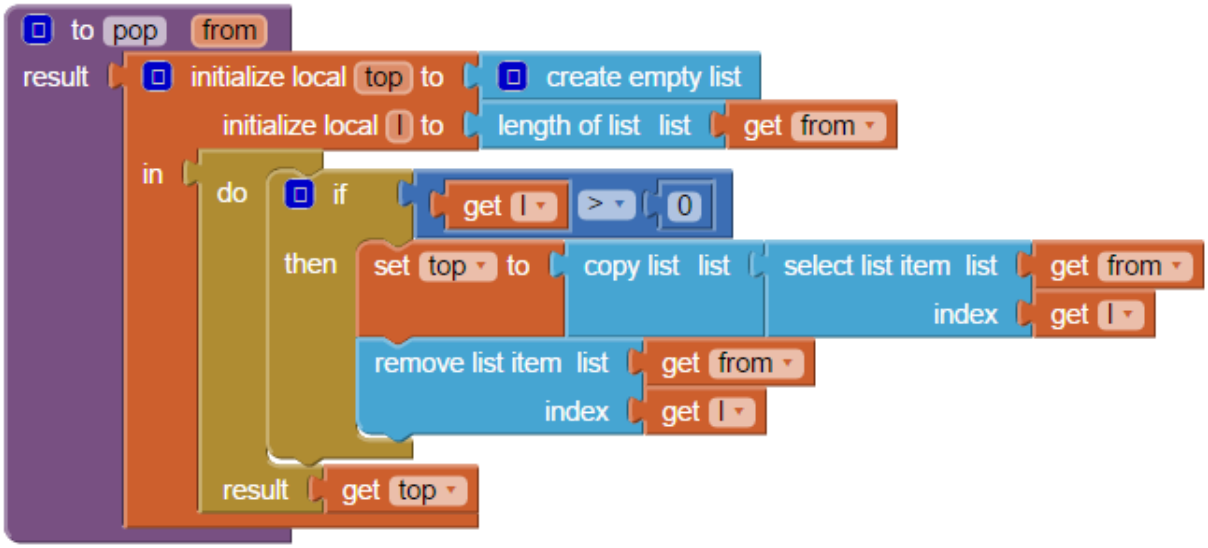
When btnUndo Clicked

The Undo button is part of a pair of buttons that let you undo moves (btnUndo), and redo undone moves (btnRedo). They work off the game table, which reflects the current state of the game through the current values of all cells, and a pair of stacks,

- Global undos
- Global redos

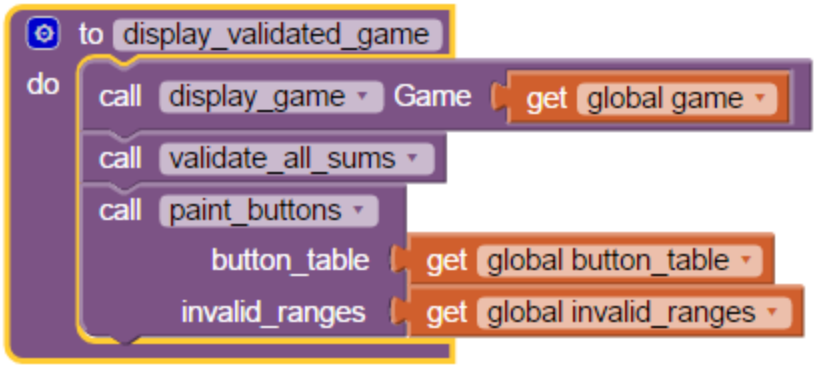
During game play, at each move a copy of the game board is pushed onto stack undoes. When a move is undone, the game board is pushed onto the redos stack, then the prior value of the game board is popped off the undos stack. Changing the board with an undo operation requires complete revalidation of the board using procedure display_validated_game.

Pop



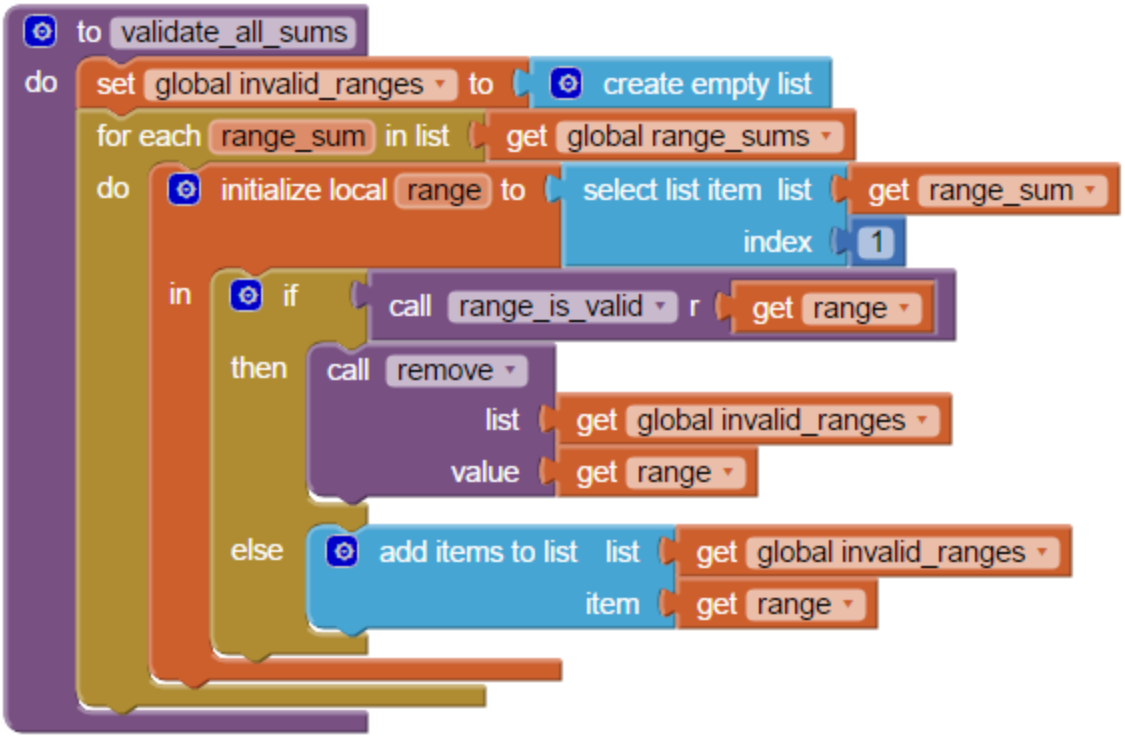
Popping from a list is done off the high index end.

Display_validated_game



We first call display_game to show the game board.
We then call validate_all_sums to check all ranges and fill global invalid_ranges.
We then call paint_buttons to highlight bad cells.

Validate_all_sums



This procedure does brute force validation on every **range_sum** pair in global range_sums.
We call range_is_valid against the **range** in each **range_sum**.
If the **range** is valid, we remove it from the global list invalid_ranges, otherwise we add it to the list invalid_ranges.

Gallery link

ai2.appinventor.mit.edu/?galleryId=5770654380064768

Other projects

https://docs.google.com/document/d/1acg2M5KdunKjJgM3Rxpy_Rf6vT6OozxdIWglgbmzroA/edit?usp=sharing