|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Text Style** | **Block Style** | **App Inventor (closest)** |
| **Assignment**   * Evaluates expression and assigns the result to the variable a. | a ← expression |  |  |
| **Display**   * Displays the value of expression followed by a space. | DISPLAY (expression) |  |  |
| **Input**   * Accepts the value from the user and returns it. | INPUT () | INPUT |  |
| **Function** | **Text Style** | **Block Style** | **App Inventor (closest)** |
| **Expressions**   * The arithmetic operators +, -, \*, and / are used to perform arithmetic on a and b. * For example: 3 / 2 evaluates to 1.5 | a + b  a - b  a \* b  a / b | a + b  a - b  a \* b  a / b |  |
| **Modulus** (Remainder)   * Evaluates to the remainder when a is divided by b. Assume that a and b are positive integers. * For example: 17 MOD 5 evaluates to 2 | a MOD b | a MOD b |  |
| **Random**   * Evaluates to a random integer from a to b, including a and b. * For example: RANDOM(1, 3) could evaluate to 1, 2, or 3 | RANDOM (a, b) |  |  |
| **Function** | **Text Style** | **Block Style** | **App Inventor (closest)** |
| **Relational Operators**   * The relational operators =, ≠, <, >, ≤, and ≥ are used to test the relationship between two variables, expressions, or values. * For example: a = b evaluates to true if a and b are equal * otherwise evaluates to false | a = b  a ≠ b  a < b  a > b  a ≤ b  a ≥ b |  |  |
| **Boolean NOT (Opposite)**   * Evaluates to true if condition is false * otherwise evaluates to false | NOT (condition) |  |  |
| **Boolean AND**   * Evaluates to true if both condition1 and condition2 are true * otherwise evaluates to false | (condition AND condition) |  |  |
| **Boolean OR**   * Evaluates to true if condition1 is true or if condition2 is true or if both condition1 and condition2 are true * otherwise evaluates to false | (condition OR condition) |  |  |
| **Function** | **Text Style** | **Block Style** | **App Inventor (closest)** |
| **Selection**  (else is optional)   * The code in the first block of statements is executed if the Boolean expression condition evaluates to true. * If the Boolean expression condition evaluates to false the code in the second block of statements is executed; or no action is taken when there is no else part. | IF(condition)  {  block of statements  }  IF(condition)  {  block of statements  }  ELSE  {  block of statements  } |  |  |
| **Function** | **Text Style** | **Block Style** | **App Inventor (closest)** |
| **Iteration** or **Repetition**   * The code in block of statements is executed n times. | REPEAT n times  {  block of statements  } |  |  |
| **Iteration** or **Repetition**   * The code in block of statements is repeated until the Boolean expression condition evaluates to true | REPEAT UNTIL (condition)  {  block of statements  } |  |  |
| **Function** | **Text Style** | **Block Style** | **App Inventor (closest)** |
| For all list operations, if a list index is less than 1 or greater than the length of the list,  an error message is produced and the program terminates. | | | |
| **Access** List Item   * Refers to the element of list at index i * The first element of list is at index 1 | list[i] |  |  |
| **Assign** List Item   * Assigns the value of list[j] to list[i] | list[i] ← list[j] |  |  |
| **Initialize** a List   * Assigns value1, value2, and value3 to list[1], list[2], and list[3] | list ← [value1, value2, value3] | |  |
| **Loop Over List**   * The variable item is assigned the value of each element of list sequentially, in order from the first element to the last element. The code in block of statements is executed once for each assignment of item. | FOR EACH item IN list  {  block of statements  } |  |  |
| **Insert** an Item in List   * Any values in list at indices greater than or equal to i are shifted to the right. The length of list is increased by 1, and value is placed at index i in list. | INSERT (list, i, value) |  |  |
| **Append** an Item to List   * The length of list is increased by 1 and value is placed at index i in list. | APPEND (list, value) |  |  |
| **Remove** Item at i from List   * Removes the item at index i in list and shifts to the left any values at indices greater than i. The length of list is decreased by 1 | REMOVE (list, i) |  |  |
| **Length** of List   * Evaluates to the number of elements in list. | LENGTH (list) |  |  |
| **Function** | **Text Style** | **Block Style** | **App Inventor (closest)** |
| **Procedure**   * A procedure, name, takes zero or more parameters. The procedure contains programming instructions. | PROCEDURE name (parameter1, parameter2, …)  {  <instructions>  } | |  |
| **Procedure** with **Return**   * A procedure, name, takes zero or more parameters. The procedure contains programming instructions and returns the value of expression. The RETURN statement may appear at any point inside the procedure and causes an immediate return from the procedure back to the calling program. | PROCEDURE name (parameter1, parameter2, …)  {  <instructions>  RETURN (expression)  } | |  |
| * **Function** | **Text Style** | **Block Style** | **App Inventor (closest)** |
| If the robot attempts to move to a square that is not open or is beyond the edge of the grid,  the robot will stay in its current location and the program will terminate. | | | |
| **Move Forward**   * The robot moves one square forward in the direction it is facing. | MOVE\_FORWARD () |  | (when using Logo app) |
| **Rotate Left**   * The robot rotates in place 90 degrees counterclockwise * i.e., makes an in-place left turn | ROTATE\_LEFT () |  |  |
| **Rotate Right**   * The robot rotates in place 90 degrees clockwise * i.e., makes an in-place right turn | ROTATE\_RIGHT () |  | (when using Logo app) |
| **Can Move?**   * Evaluates to true if there is an open square one square in the direction relative to where the robot is facing * otherwise evaluates to false * The value of direction can be left, right, forward or backward | CAN\_MOVE (direction) |  |  |