5.9 sort ................................................................. 18
5.10 modify_list ..................................................... 19
5.11 gen_random_num ................................................. 19
5.12 flip_coin .......................................................... 20
5.13 Mathematical Functions ...................................... 20
  5.13.1 sqrt .......................................................... 20
  5.13.2 power ........................................................ 20
  5.13.3 sin ........................................................... 20
  5.13.4 cos ........................................................... 21
  5.13.5 tan ........................................................... 21
  5.13.6 rollover ....................................................... 21

6 Grammar Rules .................................................. 21
ROBOT Introduction

The ROBOT programming language environment and target run-time application have been designed as a fun, intuitive sandbox with which children and novices can become exposed to, and master elementary programming concepts in the framework of a video game.

The ROBOT programming language offers a unique and enjoyable learning experience to the beginning programmer. It provides an interesting but also challenging problem domain which is intuitive to humans. The language is highly overloaded so users at all levels of exposure to math and programming concepts will be comfortable with standard operations. The compilation workflow allows for constant compiler feedback and guidance during the programming process while the visual execution environment and robot avatar gives a clear display of the program in execution.

1 IDE

The ROBOT programming language is tied to the ROBOT integrated development environment (IDE). Much like any modern IDE, we include a source tree and a window for typing code. We’ve also added a toolbox of operations and programming constructs so the user can easily browse through the language functionality in order to facilitate the novice user. Finally we’ve added a visual runtime environment, a 3D robot simulation to give the user intuitive feedback of their program in execution. The previous IDE elements are discussed further in the following sections.

1.1 Visual Runtime Environment

The robot programming language utilizes a graphical run-time environment in which the user program determines the actions (interaction with the world) of a robot avatar during a live battle simulation. Due to the nature of the language as a learning tool the graphical run-time environment is tied to the language’s IDE such that the programmer can receive constant feedback of the logical changes made to their code. The right half of the IDE’s screen space is reserved for the visual runtime environment and that is where the 3D robot simulation is displayed during execution.

While we don’t allow the A.I. program to be updated mid simulation, the visual runtime environment will still be utilized during the actual coding process. For example, each simulation will be recorded such
that the programmer will be able to replay clips of the battle or look at specific time steps. Because
the robot can display textual output, this is important feedback for logical debugging and improving
upon the A.I. In addition, when not in use, the visual runtime environment can be used to display extra
information for the programmers benefit such as helpful tips or warnings.

1.2 Programming Window

The programming window, the middle-part of the IDE, enables the user to write a program in the
ROBOT language, this window simply acts as a text editor in which ROBOT programs should be
written. When written, the program can be tested by simply clicking on the run button situated at the
top of the window. Other controls include pause and stop which will pause the simulation or have your
robot surrender respectively. Once the run button has been triggered the program will be compiled and
if the syntax is correct, the simulation will begin.

1.3 Project View

The top left window displays the brain view of the ROBOT program. The brain can be thought of
as the "project” or the collection of all code related to a single A.I. program. This always includes
the "think” routine. Similar to the main function of many popular programming languages, the think
routine acts as a starting point which the program begins at. Associated with a routine can be variables
and sub-routines. The complete view of all sub-routines and variables form a project tree of relational
memory and code.

1.4 Toolbox Window

The toolbox window is a part of the IDE which gives the main blocks of a ROBOT program such as
conditional, function declaration structure etc. When the user clicks on an item in the list, it automatic-
ally puts it into the program. This is a helpful programming device for novice programmers who are
unfamiliar with the language and require a reference manual. This way, the learning process is more
natural and well paced as users learn long or obscure keywords at their own pace and can jump into
their workflow without a steep initial learning curve. As programmers become more experienced they’ll
find it easier to simply type instructions manually and the toolbox will exist solely as a reference or for
more obscure tools they rarely use.

1.5 Toolbar

The toolbar enables the user to load, save and create new programs. It can also automatically indent
the program. While indenting, the the program is also correctly displayed. It adds the correct whitespace
and highlights the main blocks of the program. The toolbar contains the keywords which are in the
toolbox. In terms of user interface, we provides many ways to perform the same action.

2 Rules of the game/Programming objectives

For a new programmer, getting started with a general-purpose programming language and learning the
associated terminology/concepts might be a daunting task. Given that many people (kids) are already
familiar with controlling avatars in a gaming environment, ROBOT provides them with the logical step of
giving instructions to their avatar using a simple programming language instead of controlling it directly
(e.g. with a joystick).

ROBOT is built around the single idea and need for programming to be fun. The main purpose
of ROBOT is to combat the belief that programming is difficult to learn and only becomes fun and
rewarding after a long and difficult learning process. Thus ROBOT is designed to be an intuitive easy
to learn, easy to jump right into programming language framed inside the context of a video game.

The object of the game is to control a robot equipped with radar, weapons and the ability to navigate
its surroundings. Each robot has an internal energy and health meter such that if either is reduced to 0
then the robot becomes unoperational and is out of the game. The player must choose how their robot
will respond to any given situation, be it from the environment or from its opponents. As any action
requires energy the player will have to make complex decisions on how to best utilize their resources and limit their opponents. Normally the player would control the robot directly, acting as the brain of the machine deciding what the optimal moves should be. However in ROBOT the object of the game is to write programs to act as the robots brain, giving it full autonomy from you the player, who only indirectly chooses its actions.

In addition to this framework of conflict, ROBOT follows the progression of a video game through the concept of levels where players/programmers start out at the lowest levels and must defeat pre-built A.I. programs in increasingly more complex environments in order to progress in the game. Also, just as new weapons and abilities would open themselves up in a video game, new operations and programming constructs become available in more advanced program levels as they are needed. This is an intuitive and natural way to teach new programming constructs such that the player will feel excited to have a new tool with which they can use to defeat their enemy, as opposed to the framework of a class where a new operation or programming concept would feel like more work and frustrate the student.

The decision process is executed at regular intervals (interval time is optional but for this explanation we'll say every half second), while the robot is in constant motion. For example, when the simulation is initialized all robots will be stationary while they execute their first decision concurrently. Once a decision is made it will be of the form “move to location x, face towards location y, shoot”. Each robot will have its own movement speed and ability to shoot projectiles at different rates. Thus, the simulation will play out based on the current instructions. If the current instructions are completed before the next decision execution then the robot halts and waits for new instructions. If the decision process finishes before the current instructions are completed, the current instructions are replaced with the new instructions and the robot changes its behavior.

Players can customize their robots basic stats (armor, weapon fire rate, weapon damage, speed, etc.), to assist their chosen A.I. strategy. For example a robot can become a highly armored, heavy hitting tank, or a quick, more agile, rapidly firing dog-fighter. All robots have a certain number of “customization points” which can be allocated to each category of these basic stats and the distribution of these points will be editable at any point in the programming cycle.

The rules of the ROBOT game are simple, last man standing wins. However, the means by which you defeat your opponents varies greatly and are determined by your robot’s interaction with its virtual world. In order to live, flight, and die your robot has the following internal properties and mechanisms as well as opportunities in the world.

2.1 Energy

All things need energy to function, your robot is no different. Any action you take will require some amount of power from your energy reserves. Each robot has its own internal nuclear reactor which provides it a constant supply of power to its internal batteries. However, this constant supply of power is small compared to the energy needs of your robot during normal activity and can only reliably power the robots internal computers (thus running your program is the only operation the robot can do for "free"). In addition to generating its own power, extra batteries are littered across the combat arena and the robot can pick up significant one shot power bonuses which will give it a significant advantage in the middle of a fight. This requires the programmer to choose the necessary power seeking/saving strategies to defeat their opponent in a game where power is life.

2.2 Navigation

Navigation is the keystone to any proper strategy. In order to defeat an opponent the robot needs to know where the opponent is, in order to get to an opponent the robot needs to know what's between them and how to move through this environment in the best possible way. This is navigation and it’s crucial to success. Before the player can choose the best possible navigation plan, they need to know what the physical world around them looks like, luckily each robot is equipped with a "radar". The radar sends out “pings” or radar waves into the environment which bounce off its surroundings and return to it with valuable information. The robot recieves the height value of the terrain with relation to itself for a regular grid over the terrain, this is useful for avoiding cliffs and other obstacles. The robot will also receive a list of all enemies and resources within range of radar ping, this is necessary for planning a course of action in a game where information is life.
2.3 Movement

Once a plan has been made the robot needs to execute it, in order to do this movement comes into play. All robots have a base maximum speed with which they can improve through the allocation of customization points in the movement category. In order to outmaneuver one’s opponent, clever use of movement comes into play. The allocation of customization points into the movement category determines both the robot’s maximum speed and the incline in which the robot can climb a hill. Thus, not only is a more moveable robot faster, but it can also move over obstacles that less moveable robots would have to go around. Also, more moveable robots use less energy on average than less moveable robots since energy use is tied to exertion over time, so for a given energy cost for a unit of time, a faster robot can cover more distance, thus more distance can be covered for less energy. Note that energy use is not linear to movement speed. Like all things in the natural world due to friction and wind resistance, it requires more energy to maintain a faster speed than a slower speed, thus the function for energy use is $1\%$ energy per tick $\times (speed\%)^2$. In other words, each unit of time requires $1\%$ of the robot’s energy times the square of the percentage of the maximum speed the robot is traveling at. So if the robot were traveling at full speed it would use $1\%$ of its energy per tick, if it were traveling at $50\%$ of its maximum speed then it would use $0.25\%$ of its energy per tick. Keep in mind the distance a robot can travel per tick is farther for robots that have a higher max speed. Generally the robot will only want to move at full speed during direct combat and would otherwise move at a slower more efficient pace in order to reserve energy.

2.4 Attacking

Scientists in the future have created “slow light”, light which travels much slower than the traditional speed of light. Due to the nature of man this “slow light” has been weaponized into “slow lasers”, despite their obvious inferiority to conventional lasers. Each robot is equipped with one of these “slow lasers” as their primary weapon (which travel only somewhat faster than the robots themselves). Each discharge requires a significant $5\%$ of the robot’s energy and moves at a constant speed and packs a certain amount of punch based on the robot’s base weapon stats in addition to any customization points allocated to the weapon category. In general, higher weapon stats will result in lasers which move faster and do more damage. Certain powerups can give the robot a more powerful laser which is based on an “ammo” system where they only have a set amount of charges and once they’re used up the robot loses this more powerful weapon.

2.5 Taking Damage

The robot can take damage in two ways, by taking enemy fire and by falling long distances. In both cases the robot will take damage proportional to their resilience. Resilience is a base stat which can be augmented with customization points. This basic statistic determines a robot’s armor and total health. Armor can decrease the amount of damage the robot takes from enemy fire, so from a weapon of equal power, a robot with more armor will take less damage than a robot with less armor. Also, a robot with a higher resilience rating will have more health points in general. Each time a robot takes damage it loses health points, when these points are 0 the robot is destroyed and loses the match. If the robot falls of a cliff then it can potentially take damage. Falling small distances such as the robot’s own height will not cause damage to be taken, however if the robot falls more than its initial safe distance, it will begin taking damage exponentially proportional to how far it falls. Falling damage is not decreased by armor but is consistent across all robots. A higher resilience rating does however give a robot more health and thus makes them somewhat more resistant to falling then robots with lower resilience ratings.

2.6 Powerups

The robot should constantly be seeking an edge in any battle and the world offers them powerups for just such a purpose. Powerups come in the form of the three main base stats along with an energy bonus. Batteries are littered across the world which the robot can grab for a quick one shot replenishment of energy. Speed boosting powerups can also be found which adds a short rocket booster to your robot which increases its movement rating drastically for a short period of time. Weapon powerups can also...
be found which give your robot an inventory of special ammo which they can fire at the time of their choosing. These special weapons do more damage and fire faster than the regular weapon, thus each discharge is similar to a normal discharge from a weapon with a much higher weapon base statistic. Finally the last powerup replenishes lost health points and increases a robot's armor by a small amount, thus improving a robot's resilience.

3 Lexical Conventions

Like many other programming languages, ROBOT programs consists of translation units stored in files. The first thing that happens within the ROBOT runtime environment when a user executes a program is a reduction of these units to a sequence of tokens. Following this reduction, the tokens are translated into an intermediate language (i.e. Java). Finally, the user is presented with an animation to observe their avatar carry out their instructions.

3.1 Tokens

There are 5 classes of tokens in ROBOT: identifiers, keywords, constants, operators, and separators.

3.2 Identifiers

An identifier stands for the name of a variable or a function. Lexically, it consists of a sequence of characters and numbers (the underscore is considered a letter in this context). Note that ROBOT programs are case sensitive, meaning upper case and lower case letters are considered separate characters. A valid identifier starts with a letter. If the identifier is the name of a variable, it must end with a special postfix denoting its type. Valid postfixes and their corresponding types are

<table>
<thead>
<tr>
<th>Postfix</th>
<th>#</th>
<th>%</th>
<th>?</th>
<th>@</th>
<th>!</th>
<th>$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable Type</td>
<td>number</td>
<td>percentage</td>
<td>boolean</td>
<td>location</td>
<td>enemy</td>
<td>resource</td>
<td>list</td>
</tr>
</tbody>
</table>

Table 1: Postfix rules

As an example, the token closest_enemy! is an identifier for a variable called closest_enemy of type enemy.

3.3 Keywords

ROBOT programs treat the following identifiers as keywords. Accordingly, programmers are restricted from using these identifiers in any other way:

- done
- each
- else
- end
- gives
- if
- in
- instruction
- means
- self
- of
3.4 Constants
The only variable types that can be assigned or compared to a constant value in ROBOT are numbers, percentages, and booleans.

3.4.1 Number constants
Number constants can have an integer part and a decimal part. Both the integer part and the decimal part consist of a sequence of digits. The decimal part (including the period separating the decimal part from the integer part) can be omitted to represent an integer. Omitting the integer part, however, will result in a syntax error. For example, while the following two constants 0.1 and 10 are valid number constants in ROBOT, .10 is not.

3.5 Boolean constants
Boolean constants can have one of two values: true or false.

3.6 Operators
ROBOT recognizes arithmetic, comparison, and boolean operations. To improve human-readability and adoption by novice programmers, ROBOT allows textual operators as an alternative to the traditional symbolic ones.

3.6.1 Arithmetic Operators
Arithmetic operations (addition "+", subtraction "-", multiplication "*", and division "/") can be performed on numbers and percentages. Following mathematical convention, multiplication and division have precedence over addition and subtraction. Note that, multiplication and division on percentages will be converted back to percentages. Example:

\[ a\% = 100 \]
\[ b\% = 50 \]
\[ c\% = a\% \cdot b\% \]

After these 3 statements, \( c\% \) will have the value 50.

3.6.2 Comparison Operators

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Textual</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>is_strictly_lower_than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>is_lower_than</td>
</tr>
<tr>
<td>==</td>
<td>is_equal_to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>is_greater_than</td>
</tr>
<tr>
<td>&gt;</td>
<td>is_strictly_greater_than</td>
</tr>
<tr>
<td>!=</td>
<td>is_different_from</td>
</tr>
</tbody>
</table>

Table 2: Comparison operators
3.6.3 Boolean Operators

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Textual</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>not</td>
</tr>
</tbody>
</table>

Table 3: boolean operators

3.7 Separators

The only valid separators in ROBOT are parentheses, the newline character, and white space. Parentheses are used to group arithmetic and logical operations. Newlines have a special meaning as they are the only token to denote the end of a statement.

3.7.1 White Space

Blanks, tabs, and comments (described below) are considered "white space" and are grouped together as single separators between tokens. Certain characters are considered whitespace based on their location within a line. Namely, characters "|", "+", and "-" are considered whitespace when they are the first non-white space character in a line, as they are used to visually group blocks of code.

3.7.2 Comments

A comment in ROBOT begins with the characters "//" and ends at the newline character. Multiline comments are not supported (the same effect can be achieved by starting each line with "//").

3.7.3 Block Notation

ROBOT allows the programmer the use of certain characters to "draw" blocks around a set of statements that belong together (e.g. have the same scope). Consider the following ROBOT code fragment:

```robot
repeat with each enemy! in enemies!...
  enemy_loc@ = location of enemy!
  distance# = distance enemy_loc@ location@ of self
  if distance# is_less_than closest_distance# then
    closest_distance# = distance#
    closest_enemy! = enemy!
  +---done
+---done
```

In this example, the use of block notation (i.e. grouping and indentation of statements) clarifies the scope of block statement. As noted before, the characters "|", "+", and "-" are treated as whitespace since they are not preceded by any token in the lines they appear in. Because of their treatment as whitespaces, the use of the block notation is optional and is left up to the programmer. However, the usage of block notation could be enforced/automated with the help of an IDE, resulting consistent and easily comprehensible code across developers.
4 Meaning of Identifiers

4.1 Variables

4.1.1 Primitive Variable Types

ROBOT features 3 primitive variable types: number, percentage, and boolean.

**Number**  Robot supports floating point numbers only. Integers can be assigned to number variables, but internally they will be stored as floating point numbers.

```
// setting an integer and a decimal number variable
num1# = 10
num2# = 0.1
```

**Percentage**  Robot has support for variables which can support percentages. These are required because ROBOT frequently has to deal with variables which represent the health of the robot or the energy level of the robot. It is intuitive to have a percentage value define such a parameter. A percentage variable is constrained to have a value between 0 and 100 (both inclusive).

```
// setting a percentage variable
health% = 50
```

The value of a percentage variable must be between 0 and 100 (inclusive). For example, adding 70% and 40% is capped at 100%. Similarly, the subtraction operation is capped at 0%.

```
health_boost% = health_boost% of resource$ // health_boost of this resource is 70%
health% = health% + health_boost% // Health is now 100%
```

When a percentage variable is used in an arithmetic operation with a numeric variable, it is treated as its numeric equivalent (for example multiplying 50% with 250 is the same as multiplying 0.5 with 250).

A numeric value can be stored in a percentage variable and vice versa. Conversion between these types is taken care of by the ROBOT translator. However, the user should watch out for the clipping that may occur when storing a number in a percentage variable.

```
// multiplying a percentage with a number and storing it as either a percentage or a number
multiplier# = 2
health% = 60
healthNew% = health% * multiplier# // healthNew is now 100% due to clipping
healthNum# = health% * multiplier# // healthNum is 1.2
```

**Boolean**  The other basic data type supported by ROBOT is boolean. A boolean variable can be either true or false.

```
// setting a boolean variable
attack? = true
```

Boolean variables can be used together with boolean operators, boolean constants, and parentheses separators to build arbitrarily complex boolean expressions.

4.1.2 Objects

Robot supports 3 built-in object types that can store other objects or primitives as attributes of a logical object: location, enemy, and resource. The attributes of an object can be accessed using the of expression (see below for examples).
**Location**  The location object stores the 2D cartesian coordinates of a given location on the game map. Additionally, it also stores the height of that location (i.e. determines cliffs and hills). It has 3 numeric attributes \( x \), \( y \), and \( h \).

```plaintext
// accessing x and y attributes of a location variable
x = x of location@
y = y of location@
h = height of location@
```

**Self (implied)**  The avatar that the user program controls is represented by the implied self object. Even though the user never has to store the self object in a variable, the attributes of the self object are available to the user through the use of the `self` expression, which is a special case of `of` expressions. The self object has the following attributes:

- location@
- health%
- energy%
- enemy_info!
- resource_info$
- ammo#

```plaintext
// accessing the health attribute of the self object
current_health_level% = health% of self
```

**Enemy**  The enemy object has a subset of the attributes available in the self object. Conceptually, this follows from the fact that both the self object and the enemy objects represent robots within the game, but the user has access to more of attributes available to their own robot.

- location@
- health%

```plaintext
// accessing the location attribute of an enemy variable
enemy_loc@ = location@ of enemy1!
```

**Resource**  The resource has attributes to indicate its location on the map, and numeric attributes that express what a robot would gain from acquiring this resource in terms of ammo and health.

- location@
- health_boost%
- ammo_stash#

```plaintext
// accessing the ammo_stash# attribute of a resource variable
additional_ammo# = ammo_stash# of resource1$
```
4.1.3 Lists

Robot supports grouping of variables or constants in lists. Even though there is no interface for a programmer to create or modify lists, they can be returned by built-in functions, stored in list variables and iterated over to access each element. Each list in ROBOT is assigned a type at the time of declaration. Lists that contain multiple types are not permitted in ROBOT.

```
// let's iterate over the list of enemies
repeat with each enemy! in enemies!...
  | enemy_loc@ = location of enemy!
  | distance# = distance enemy_loc@ location@ of self
  |---done
```

Programmers can also access individual elements within a list using the th expression.

```
enemy2! = 2th of enemies!... // assigns 2nd element from enemies!... to enemy2!
```

Lists cannot be nested (i.e. a list cannot contain another list as one of its elements).

```
min_damage% = 100;
myloc@ = location@ of self
loc_with_min_dam@ = NOWHERE@
repeat with each dir@ in directions@... // directions is a global list with 8 unit vectors
  | // N, NE, E, SE, S, SW, W, NW
  | temp_loc@ = myloc@ + dir@ // We should define adding 2 locations
  | temp_damage% = calc_damage myloc@ temp_loc@ // height is stored in loc@
  | if temp_damage% is_less_than min_damage%
  |   | min_damage% = temp_damage%
  |   | loc_with_min_dam@ = temp_loc@
  |   |---done
```

4.2 Functions

ROBOT programs consist of a series of instructions to the robot avatar. These instructions can be grouped together into logical computational units known as functions. While some of these functions return a value after they complete their calculations, others simply run the enclosed computation without returning anything. Some of the robot's low-level functionality is provided to ROBOT programmers via a built-in standard function library, but they are encouraged to write their own functions to describe arbitrarily complex game playing logic and to be able store and re-use portions of their code.

The syntax to define a function without a return value is

```
instruction function_name (with arguments1 ... argumentN) means
  | (statements)
  |---done
```
Optional arguments can be given following the function name with the keyword with and a list of arguments separated by whitespaces. Functions without return values are also known as subroutines.

The when a function has a return value, it is specified by using the following syntax:

\[
\text{instruction function_name (with arguments1 \ldots argumentN) means } \\
\quad (\text{statements}) \\
\quad \text{gives return_value[!@#$\ldots]}
\]

The return value of a function can be any supported ROBOT data type.

One special subroutine is the think subroutine. The think subroutine is entry point to a ROBOT program and as such, every ROBOT program must include a think subroutine somewhere in their code. The syntax for the think subroutine is similar to a function with no arguments and no return value:

\[
\text{think} \\
\quad (\text{statements}) \\
\quad \text{done}
\]

Alternatively, the think routine can also be thought of as the "think loop", since the game play consists of the robot avatar running the think subroutine iteratively over the duration of the game.

### 4.3 Scope of Identifiers

Similar to other programming languages, each identifier defined in a ROBOT program has a scope, a region of the program where it’s known.

Variables declared outside any functions (including the think loop) are global in terms of their scope. These variables maintain their value over each iteration of the think loop.

Variables declared inside a function will be only visible while that function is running will be erased from memory as soon as the function completes executing. Similarly, any variable defined within an even more granular code block, such as an if-then statement or a repeat-loop will be local to those blocks and they will be destroyed after the conditional / after each iteration of the repeat-loop.

Since instructions, repeat-blocks, and if-then-blocks can be terminated by different keywords (e.g. end, done, gives), the best way to keep track of variable scope is to make use of the visual cue provided by the block notation. The block closest to a variable will enclose the portion of the code where that variable is visible/available.

### 4.4 Loops

A ROBOT program can contain one or more of the supported looping forms. There are three basic loop constructs supported by ROBOT.

#### 4.4.1 repeat with each

The first looping construct is the repeat with each construct which can be used to iterate over all the elements of a list and possibly do some processing on each element of the list. A simple example of this construct:

\[
\text{repeat with each dir@ in directions@\ldots} \\
\quad // \text{process dir@} \\
\quad \text{done}
\]

Here repeat, with, each, in and done are keywords. dir@ is a variable which contains the current element of the list represented by directions@\ldots variable. So the dir@ variable takes the value of the first location in directions@\ldots on the first iteration, second location in directions@\ldots in the second iteration and so on. So if directions@\ldots has location1@, location2@ and location3@ in that order then dir@ will
equal location1@ in the first iteration, location2@ in the second iteration and location3@ in the third and final iteration. Since there are only three elements in the list, the loop will terminate after iteration 3.

4.4.2 repeat times

The second looping construct available in ROBOT is called the repeat times construct. It is intuitive to use this construct when we want a certain set of statements to be repeated a fixed number of times. The syntax of this construct is as follows:

```
repeat 5 times
| // statements to be repeated 5 times
+---done
```

Here repeat, times and done are keywords. The loop in the above example will execute the statements it surrounds five times. We can use a number variable instead of hard coding a number as is done by writing '5' in the above snippet. If a number variable is used instead and that number variable contains say 5.65, the decimal part would be truncated and 5 iterations of the loop will take place. So the number of times a repeat times loop can iterate must be a positive integer. If the number variable contains a value less than one, then the loop won’t execute at all.

4.4.3 while

A while loop construct in ROBOT is very similar to a while loop in other languages like C and Java. It is intended to be used where we execute a set of statements until a boolean condition/variable is true. The syntax of this looping construct is as follows:

```
while not rain? or sunny?
| // statements which modify rain? and/or sunny?
+---done
```

Here while and done are keywords. The above loop executes the statements it encloses until the boolean condition which follows while is true at which point the looping terminates. You can write and valid any boolean expression after while.

4.5 Conditional execution

ROBOT supports if conditional constructs. The syntax of the if is as follows:

```
if temp_damage% is_less_than min_damage%
| // do something
+---done
```

Here if, is_less_than and done are keywords. The above construct will cause the boolean condition following the if to be evaluated. If the boolean expression evaluates to true then the statements surrounded by the if construct are executed; otherwise the control skips to the first statement following the if construct.
5 Standard function library

5.1 say

Argument(s):

- string literal
- $>identifier<$

Returns: nothing

The say function is equivalent to the print/write function in traditional programming languages. The arguments to the say function will be displayed on the game screen as a speech balloon over the robot avatar’s head.

String literals do not have to enclosed in quotes. Any alphanumeric character following between say and the end of the line will be treated as part of the string literal. As an exception, characters enclosed between $>$ will be treated as identifiers. In those cases, the say function will print the value of the variable instead of the identifier itself.

Example:

say Oh, oh. My health is now $>health\%$ of self$<$

5.2 move_to

Argument(s):

- location@$$
- speed$

Returns: nothing

Cost: 1% energy per tick $* (speed\%)^2$

The move_to function takes a location object and a speed percentage as its sole arguments and causes the robot avatar to start moving from its current location to the location specified in the argument at the given percentage of its maximum speed. Since avatar movements are governed by the physics of the virtual robot world, move operations are not instantaneous and are carried out over a duration (calculated by the game engine based on the robot’s maximum speed and the distance to be travelled). Like all physical actions that can be undertaken by the robot, move operations result in a decrease in energy which must replenished before it reaches 0%. Note that subsequent think cycles executed after the initiation of the move operation may result in the robot deciding on a new course of action based on new input and break up its travel to carrying out a new task. The energy cost to move is based on the speed at which the robot moves. Each robot can have its own customized maximum speed based on the allocation of customization points. The energy cost associated with movement is based on a given percentage per tick (unit of time) times the square of the speed the robot is moving at. In other words, at 100% or maximum speed the robot will use 1% of its energy per meter. At 50% speed the robot will be using 0.25% of its energy per meter. The energy to speed relation is exponential rather than linear.
5.3 shoot

Argument(s):

- enemy!
- using_powerup (optional)

Returns: nothing

Cost: 10% energy per rocket

Additional Resource Requirements: ammo > 0

The shoot function results in the robot firing a laser towards the enemy passed as the argument to the shoot function. Since there is delay between the launching of the laser and the laser reaching its target, it is possible for the enemies to maneuver out of the path of a beam and avoid taking the associated damage. Obviously, the programmer should ensure that the robot has the necessary energy, otherwise the robot will waste its energy and become an easy prey for its enemies. Certain powerups can give you extra ammo energy. This results in more powerful laser blasts which do not require normal energy to use, thus an ammo counter keeps track of these extra ammo batteries and they can be discharged at any time to add extra firepower in the heat of a battle.

Example:

```plaintext
// this example illustrates how to check if a laser powerup is available and how to use it
if ammo of self is_greater_than 0 then
  shoot enemy! using_powerup // Powerup laser: this decreases ammo by 1, no energy is used
else
  shoot enemy! // Regular laser: this decreases energy by 5%, no ammo is used
end
```

5.4 ping

Argument(s): none

Returns: nothing

Cost: 5% energy per ping

The ping function causes the avatar to ping its surrounding using its radar. This ping helps the avatar get up-to-date information on the whereabouts of all enemies and resources present in the world at that time even beyond the robot’s visible area (squares within 5 steps in each direction). These updated locations are available to the robot via the built-in lists: enemy_info!... and resource_info$.... The programmers aim should be to use radar pings with an optimal frequency to avoid traveling to locations where enemies/resources used to be long time ago, yet saving enough energy to win in a combat situation once the enemies are engaged.
5.5 distance
Argument(s):
  • location1@
  • location2@
Returns: distance#

The distance function returns the cartesian distance between 2 squares in the map.

5.6 get_enemies
Argument(s): nothing
Return(s): enemies!

Returns the list of all enemies on the map.

5.7 get_resources
Argument(s): nothing
Return(s): resources$

Returns the list of all resources on the map.

5.8 get_environment_height
Argument(s):
  • location@
Return(s): height#

The get_environment_height function takes as input a location@ argument and returns the height value of the environment for that position. The idea behind the get_environment_height function is that the robot is aware of the terrain surrounding it from the beginning of the simulation and is able to access the information through the get_environment_height function, thus enabling the robot to do a virtual walkthrough of the environment in its mind and thus choose the best path to avoid obstacles such as walls and cliffs when moving in the actual battle.

5.9 sort
Argument(s):
  • list[#%!@$]...
  • direction: < or > (optional, default: <)
  • attribute (optional, default: location)
Returns: sorted_list[#%!@$]...

The sort function takes a list and returns a new sorted copy of that list. In the simplest case, when the sort function is called with a list that contains either numbers or percentages, it sorts that list in an ascending order based on the numeric value of each element. The default sort order can be either overridden by using > or explicitly stated by specifying <. If the list contains enemies, resources or locations, the default behavior of sort is to order the elements of the list based on the distance of each element to the current location of the robot. Again, the default behavior can be overridden by specifying an attribute of the type of the elements in the list (see below for examples).

Example(s):
// sorts a list of numbers with default options (ascending, based on numeric value)
sorted_list... = sort list_of_numbers#
// sorts a list of enemies in a descending order based on default sort attribute (i.e. location)
furthest_enemies!... = sort enemy_info!...
// sorts a list of resources based in a descending order based on a custom sort attribute
most_valuable_resources$... = sort resource_info$... > ammo_stash#

5.10 modify_list
Argument(s):
- list[#%!@$]...
- add or remove
- item[#%!@$] or item_at
- index# (optional)

Returns: nothing

The modify_list function let’s the programmer add elements to/remove elements from a list. The
default add operation (without specifying the optional argument index#) adds an element at the end
of the list. When the optional index is specified, the add operation inserts the element at that position
and increases the index of any elements past that position by one. On the other hand, the default
remove operation (without specifying the optional argument index#) searches the list for the item given
as argument #3 and removes it from the list (if found). The index of any elements past the position
of the deleted element are decreased by one. Last but not least, rather than removing a given element,
the programmer can choose to delete the item at a specific position in the list, by calling passing the
optional index argument as the 4th argument to modify_list following these 3 arguments: list name, remove, and the item_at place holder.

The following examples demonstrate what arguments to pass to the modify_list function to get the
different behaviors mentioned above.

Example(s):

// adds loc_of_cliff@ to the end of cliff_locations@
modify_list cliff_locations@ add loc_of_cliff@
// inserts loc_of_cliff@ as the 3rd element in cliff_locations@
modify_list cliff_locations@ add loc_of_cliff@ 3
// removes loc_of_cliff@ from cliff_locations@ (if it is present)
modify_list cliff_locations@ remove loc_of_cliff@
// removes 3rd item from from cliff_locations@
modify_list cliff_locations@ remove item_at 3

5.11 gen_random_num
Argument(s): none

Returns: randomNumber%

This function picks a random number from a uniform distribution between 0 and 1 and returns it as
a percentage. It can be used to introduce probabilistic aspects into the think routine rather than making
every decision deterministic.
5.12 flip_coin

Argument(s):
- heads% (optional, default: 50)

Returns: heads?

The flipCoin function can be called without an argument to return a random boolean value with equal probability of being true or false. Similar to the previous function, gen_random_num, this function can be utilized to introduce probabilistic aspects into the think routine. Additionally, the fairness of the coin can be adjusted by passing a percentage value which scales the likelihood of the function returning true.

Example(s):

// if health is 75%, the likelihood of should_attack? being true after this statement will be 3/4
should_attack? = flipCoin health% of self

5.13 Mathematical Functions

ROBOT provides programmers with an interface to a few common mathematical functions that can be used within ROBOT programs for implementing advanced logic

5.13.1 sqrt

Argument(s):
- number# or numeric expression

Returns: number#

Returns square root of the numeric value passed as its argument

5.13.2 power

Argument(s):
- number# or numeric expression
- exponent# or numeric expression

Returns: number#

Raises the 1st argument to the power indicated by the 2nd argument and returns the resulting value as a number

5.13.3 sin

Argument(s):
- angle#

Returns: number#

Returns the sine of the angle passed as the argument. The angle passed as argument must be specified in degrees.
5.13.4 cos
Argument(s):
  • angle#
Returns: number#

Returns the cosine of the angle passed as the argument. The angle passed as argument must be specified in degrees.

5.13.5 tan
Argument(s):
  • angle#
Returns: number#

Returns the tangent of the angle passed as the argument. The angle passed as argument must be specified in degrees.

5.13.6 rollover
Argument(s):
  • number1#
  • number2#
Returns: number#

Returns the modulus of number1# with respect to number2#. number = number1 mod number2. They can be non-integer since we do not differentiate number types.

6 Grammar Rules

Below is the grammar of the ROBOT language. The grammar has undefined terminal symbols: number_expression, constant_expression and string. A number_expression is an element of \([0-9]^+\) and a constant_expression is one of \([a-z][a-z0-9]*\) which is not a keyword. A string is created from any character except < and >. The “one of” statement means that the symbols are terminal.

program:
  declaration entry functions

declaration:
  variable_name \n declaration
  field_access assign_op value \n declaration
  epsilon

field_access:
  variable_name of field_access
  index of field_access
  variable_name
  self

index:
  number order
order: one of
   st nd th

assign_op: one of
   = is

value:
   variable_name
   base_value
   boolean_expression
   arithmetic_expression
   constant_expression of variable_name
   constant_value

constant_value: one of
   MAX# MIN# NOTHING! NOWHERE@

base_value:
   boolean_final
   number_expression

arithmetic_expression:
   number_identifier arithmetic_operator arithmetic_expression
   number_identifier

arithmetic_operator: one of
   + - / *

entry:
   think \n statement end

statement:
   call
   declaration
   loop
   conditional
   epsilon

call:
   function_name argument \n
say say_argument
sort list_name sort_argument

say_argument:
   string variable_string say_argument

variable_string:
   > variable_name <
   epsilon

sort_argument:
   first_sort_option second_sort_option
first_sort_option: one of
  < >
  epsilon

second_sort_argument:
  variable_name_without_list
  epsilon

function_name:
  constant_expression

argument:
  value argument
  epsilon

loop:
  repeat number_identifier times
  statement done
  
  repeat with each variable_name in list_name
  statement done
  
  while boolean_expression
  statement done

number_identifier:
  number_name
  number_expression

conditional:
  if boolean_expression then
  statement else_statement
  done

else_statement:
  else
  statement
done
epsilon

variable_name:
  number_name
  list_name
  boolean_name
  percentage_name
  location_name
  enemy_name
  resource_name
  list_access

list_access:
  number_identifier of list_name

boolean_expression:
  boolean_expression boolean_operator boolean_expression
  not_operator boolean_expression
  (boolean_expression)
  boolean_name
  boolean_final

not_operator: one of
  ! not

boolean_final: one of
  true false
boolean_operator: one of
== is_equal_to <= is_lower_than < is_strictly_lower_than > is_strictly_greater_than
>= is_greater_than != is_different_from and && or ||

number_name:
  constant_expression#

list_name:
  variable_name...

boolean_name:
  constant_expression?

percentage_name:
  constant_expression%

location_name:
  constant_expression@

datatype_name:
  constant_expression!

resource_name:
  constant_expression$

variable_name_without_list:
  number_name
  boolean_name
  percentage_name
  location_name
  enemy_name
  resource_name
  list_access

function_name:
  constant_expression

functions:
  function \n functions

function:
  instruction function_name with_statement means \n statement gives return_value

with_statement:
  with arguments
    epsilon

return_value:
  value
    nothing