# Contents

Introduction 3

1 Playing the Game 3
   1.1 Energy 4
   1.2 Navigation 4
   1.3 Movement 4
   1.4 Attacking 5
   1.5 Taking Damage 5
   1.6 Powerups 5

2 Getting Started 6

3 Types, Operators and Expressions 7

4 Control Flow 8

5 Functions and Program Structure 9

6 Sample codes 11

Make ROBOT your own 13
Introduction

Let us begin with a quick introduction to ROBOT. Our goal is to convey the essential elements of the language by showing ROBOT in action through real programs. In order to grow the readers understanding of ROBOT we’ve decided to format this tutorial in such a way that we start with the simplest program possible in the ROBOT language and introduce features one at a time on a "need to know" basis such that the reader will not get bogged down in details, rules and exceptions. In fact ROBOT is designed such that the user should never be bogged down in such things and that the user should be able to write complete programs to control their robot at any experience level. This tutorial however will give ample exposure to variables, constants, arithmetic operations, control flow, subroutines, and the details of input and output as well as the concept of the ROBOT game and the rules and mechanisms by which the programmer can win.

1 Playing the Game

The ROBOT programming language is in essence a video game. 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).

The ROBOT video game 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.

When playing ROBOT, you’ll be given control of your own robot combatant who lives in a small virtual world consisting of a terrain with possible walls and cliffs as well as resources and power-ups scattered across the landscape. Other enemy robot combatents also inhabit your small virtual world and you must defeat or outlive them in order to win each match. Each robot is 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 non-operational 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 order to progress to higher levels, you must face and defeat increasingly smarter foes as well as master more advanced and complex programing concepts. ROBOT starts out simple on a flat terrain with a single robot opponent and limited programmability. However, as the levels progress you must fight across trecherous terrains with an ever increasing toolbox of functions and operations to choose from. Just as the terrain and opponents become more difficult, power-ups become available to assist the player or opponent in getting an edge in battle.

Although your robot is constantly in motion, it thinks between short intervals. At each think step your program is executed, a decision is made, and the robot carries out that decision for the given time unit until the next think cycle. 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
behaviour.

Your robot has three basic statistics associated with it. Movement, weapons and resilience. At any
point during the programming process before each match, you can allocate a certain number of ”cus-
tomization points” into each of these three categories. The more points you assign to a given category
will increase your robots strength in that category. It is wise to augment your base statistics in order
to assist your chosen A.I. strategy. For example a robot can become a highly armored, heavy hitting
tank, or a quick, agile, rapidly firing dog-fighter. The strategy you choose should determine how your
customization points are used.

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. Once you have a solid understanding of the following robot char-
acteristics and game mechanics, you should be ready to start writing your first ROBOT program! Good
luck and happy hunting!

### 1.1 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 you to choose the necessary power seeking/saving strategies to
defeat their opponent in a game where power is life.

### 1.2 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 is 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 robot can choose the best possible navigation plan, it needs to know what the
physical world around it 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 with valuable
information. The robot receives 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.

### 1.3 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 ones oppenent, clever use of
movement comes into play. The allocation of customization points into the movement category determines
both the robots 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 \* (speed\%)^2. In other words, each unit of time requires 1% of the robots 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.

1.4 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 robots energy and moves at a constant speed and packs a certain amount of punch based on the robots 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.

1.5 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 robots 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 resistance ratings.

1.6 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 robots armor by a small amount, thus improving a robots resilience.
2 Getting Started

Respecting tradition we begin our introduction to ROBOT with the standard "hello, world" program. This initial example, although simple, marks a huge success for the beginning ROBOT programmer as it gives them direct control of their robot and successful execution means they’ve cleared the hurdle of getting the ROBOT IDE up and running and executing a program in it. The ROBOT "hello, world" program is

```
[Sample code 1 : Hello world]
1    think
2   |    say hello, world
3    ---end
```

The process of getting ROBOT up and running is easier than in other languages due to the integrated development environment tied to the run-time environment. In order to truly get started and write the "hello, world" program for yourself, we’ll quickly introduce the ROBOT IDE and explain how to use it. After starting the ROBOT IDE the first thing you see are four main windows, two skinny windows on the far left of the screen one on top of the other, and the remaining screen space divided into two more larger windows. These four windows allow the user to interact with and view all information tied to their ROBOT program.

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 window below contains a list of all operators available to the user at any given time. The idea behind ROBOT is that it’s highly overloaded and should be easy to use. In order to achieve a highly overloaded design while keeping explanations and the amount of information the user needs to remember small, we added the toolbox window which contains all commands, operations and implicit variables in a tree structure such that the user can simply search for their operations of choice and quickly see which operations are duplicates. The toolbox window should at first be heavily relied upon when starting out, but as the user becomes more comfortable and experienced with ROBOT the toolbox should become a reference material more than an integral part of the programming process.

The middle window is arguably the most important as it contains the text editor in which the user writes their ROBOT programs while the right hand window is the 3D visual runtime environment. Above the program editor are the start, pause, and stop buttons, intuitively these compile and run the program, pause its execution and terminate it respectively.

After writing the "hello, world" program, press the start button and watch your robot speak to you in the runtime visualizer. With your first ROBOT program up and running you’re now ready to jump into variables, operators and expressions!
3 Types, Operators and Expressions

Now that you’ve gotten your first ROBOT program up and running, let’s see how to make your robot move to a new location in the world. This example makes use of variables of multiple types as well as the assignment operator and basic use of expressions.

[Sample code 2: Move Robot]

```
think
1    | x# of targetLocation@ = 10
2    | y# of targetLocation@ = 10
3    | move_to targetLocation@ 100%
4    +--end
```

Three variable types are used in this example. A variable is in essence just data identified by a name. The kind of data which a variable holds is determined by its type. In ROBOT the type of a variable is denoted by a special character at the end of the variables name. In this example we see the variable types # (number) and @ (location) and % (percentage). The # type can hold any real number, that is, any number, positive or negative with an integer and decimal component. The % type is a percentage from 0-100%, in essence a % variable is a # which can only be assigned the subset of real numbers between 0 and 1 inclusive. The @ type is a location value on a two dimensional plane. That is, the @ variable in essence stores two # variables known as x# and y#. ROBOT supports several primitive data types, # (numbers), ? (booleans), and % (percentage). The rest of the data types supported by ROBOT can be thought of as complex data types, or objects if you’re familiar with Java or C++. A complex data type can contain several primitive or complex variables. As we just stated the # variables are primitive data while the @ variable is complex data since it consists two primitive variables.

In line 2 of the sample code we see the line `x# of targetLocation@ = 10`, the first variable in this expression is x#, a variable which stores a real number. Next we see the "of" keyword, which means that x# is a member of the following complex variable. We see that the complex variable following the of keyword is of the type location which in fact does contain the variable x# thus we’ve successfully accessed a chunk of data for reading or writing. The assignment operator = means that whatever value is read from the right hand side of the operator, is written to the variable on the left hand side of the operator. In this case the x# value of our location object is assigned to the value of 10. After the assignments performed in lines 2 and 3, targetLocation@ has the two dimensional vector [10, 10].

In line 4 the `move_to` function is called with the `targetLocation@` and 100% parameters passed to it. `move_to` is a function in the ROBOT standard library, a collection of useful functions which will assist you in computation and controlling your robot. The `move_to` library call takes a location variable in global coordinates as well as the percentage of the robot’s maximum speed at which it should move. In this example the robot would be instructed to move from wherever it is to the location [10, 10] at its full speed. After calling this function the robot will continue to move to this location at this speed until either its given a new location or speed, it’s destroyed, it runs out of power, or it arrives at this location in which case it will stop.
4 Control Flow

Next we introduce control flow and give a quick example how to make your robot walk in a square pattern. Several new programming concepts are introduced here, we see our first use of lists and conditional statements.

Sample code 3 : Patrol Robot

```python
1 count# = 0

2 think
3     | my_loc@ = location@ of self
4     |
5     | x# of 1th of targetLocation@...=10
6     | y# of 1th of targetLocation@...=y# of my_loc@
7     | x# of 2th of targetLocation@...=10
8     | y# of 2th of targetLocation@...=10
9     | x# of 3th of targetLocation@...=x# of my_loc@
10    | y# of 3th of targetLocation@...=10
11    | x# of 4th of targetLocation@...=x# of my_loc@
12    | y# of 4th of targetLocation@...=y# of my_loc@
13    |
14    | if my_loc@ != count#th of targetLocation@...
15    |     | move_to count#th of targetLocation@...
16    |     | | +---else
17    |     | | count# = (count# + 1) rollover 4
18    |     | +---end
19    +---end
```

Line 2 introduces two new keywords, `self` which allows you to identify your robot and access its data, as well as `location@` which refers to your robot’s location. In lines 4-11 we see the creation of a list of four location variables. Lists can be used for any data type, primitive or complex and are identified by the "..." following the variables name and type identifier. A list is not a fundamentally alter a variables data type, but instead denotes a collection of variables of that data type. If a list is used without an index location (an identifier to the specific element in the list), then the 0th element in the list is used by default. In this case we see the "th" keyword used to denote a specific element in the list. The of tag is of course still used to give relations. In this case we see that our x# and y# are of a specific element of our location list. From wherever we randomly start on the map, to [10, 10], we now have 4 location variables which produce the 4 corners of a rectangle including those two points.

Line 13 introduces our first conditional statement, the `if`. Conditional statements take as input a boolean value and depending on whether that value is true or false, they run or skip segments of code. In this case we check if our current location is the same as our target location. If it is not then we are still on our way to the target location and we are simply instructed to continue moving to that point. If we have reached our target location then the conditional statement will be given a value of false since `location@of self` and the current target location will equal each other. If this is the case then we branch to the else statement and run the instructions specified there. Note that not all if statements require an else term, it is simply a convenience to the programmer who does not wish to write two identical if statements with opposing truth values. The count variable is declared outside of the think loop, this denotes that count is a gobal variable and persists between thinks. That is, if say the value 3 is assigned to count in one think cycle, then in the next think cycle count will still equal 3 where all the local variables declared inside the think block will be empty. In this example our global count variable is used to index the current desired target location. When one location is reached the count variable is incremented so in the next think cycle the new location will be used. We see on line 20 how the count variable is incremented. Make note of the rollover system library function, this is identical to the mathematical modulus. The value on the left is modulated by the value on the right, thus when our
count variable is greater than 4, we remove 4 repeatedly from our count until it is no longer greater than 4.

5 Functions and Program Structure

Now let's move on to programmer defined functions and their use in the think block.

[Sample code 4 : Find and shoot enemy]

1  think
2   |
3   | closest_enemy! = NOTHING!
4   | // 1. Get enemy locations
5   | ping_radar // Decreases energy
6   | enemies!... = get_enemies // alternatively: enemies!... = enemies!... of radar
7   |
8   | // 2. Find closest
9   | closest_enemy! = find_closest_enemy enemies!...
10  |
11  | // 3. Shoot
12  | shoot closest_enemy! // Decreases energy
13  |
14  +---end
15
16  instruction find_closest_enemy with enemies!... means
17   |
18   | closest_distance# = MAX#
19   | closest_enemy! = NOTHING!
20   | my_loc@ = location@ of self
21   |
22   | repeat with each enemy! in enemies!...
23   |
24   | | enemy_loc@ = location of enemy!
25   | | distance# = distance enemy_loc@ my_loc@
26   |
27   | | if distance# is_less_than closest_distance# then
28     |
29   | | | closest_distance# = distance#
30   | | | closest_enemy! = enemy!
31   | | |
32   | | | +---done
33   |
34   | | +---done
35   |
36   +---gives closest_enemy!

We introduce the enemy variable type "!". Whereas enemies are robots just like you, in terms of gameplay you have less information about them than you do for yourself, thus any given enemy can be computationally stored as an enemy variable. The enemy data type is a complex data type which stores a location@ for where the enemy is at in the world as well as health# and energy%. In line 3 we declare an enemy variable which we plan to store the closest enemy in. We initialize it to nothing, as in there is no enemy.

Line 5 gives us our first look at the radar and our means of which we can make decisions with. Each robot contains a radar type. The radar is not given a special character to denote its type because its a special complex object which can only be read from, not written to, also it is implicit to the robots type itself and there can be only one, thus it is not necessary to distinguish it from other variables in terms
of type because the word radar itself is a distinguishing factor. In line 5 we call the library function `ping`. This tells our radar to search the surroundings at this moment and gather a list of all the enemies and resources within range. This operation requires energy, if the robot does not have enough energy in reserves then the ping will fail and the robot's energy will be reduced to 0. Ideally an experienced programmer would use an if statement to check the amount of energy their robot has in reserves and they would only ping if they had the energy to do so.

Line 6 makes use of the `get_enemies` library call which returns a list of all enemies detected by the radar, with data being up to date with the moment they were pinged. The convenience of the radar's list is that it identifies enemies and resources so upon future pings, if the same enemy is detected again, rather than adding a new enemy to the list and keeping the old outdated information, it will recognize that it has found the same enemy as before and update its information. Alternatively we could access the radar's internal enemies list, however this is generally less safe and considered bad programming practice.

Line 9 makes use of our function `find_closest_enemy` defined on line 16. Given a list of enemies our function simply searches through them and returns the enemy closest to the player. Line 12 then instructs the robot to fire its laser at the closest enemy just found, again using up energy. This simple program is an example of a step in the attacking process by which the robot once deciding it would like to attack, looks for enemies, finds one, and then attacks it.

Our function on line 16 structurally looks very similar to our think block, however it starts with the instruction keyword which tells ROBOT that we are creating our own instruction. Next we see the new instructions name and the data that it would like to input. The means keyword follows instruction and denotes that we are finished declaring what our instruction will be named and what data it requires. At the very end of the instruction block we see the gives keyword, just as instructions may take variables as input, they can also give variables as output. The difference is that an instruction can take many inputs but can at most ever have one output, here we see that we are returning an enemy object. Lines 18-20 initialize variables in the same manner we've already seen. Line 22 introduces our first loop, the `repeat with each` loop goes through an entire list and for each element in that list, repeats the code given in its block, we see that `done` fences off the end of this code. In this case we are looping through each enemy in the enemies list. For each enemy we get its location, find the distance from our robot to the enemy using the distance library call, then decide if our current enemy is closer than our current closest enemy, then replace our closest enemy with the enemy currently being checked. Once this list is completely iterated through, we should have our closest enemy which we then give as output from the instruction back on line 9.
6 Sample codes

Much like the previous example, here we attempt to find the closest resource instead of enemy and then move to it.

[Sample code 5 : Find resources]

```
1   think
2   |
3   |  closest_resource$ = NOTHING$
4   |  // 1. Get resource locations
5   |  ping // Decreases energy
6   |  resources$... = get_resources
7   |
8   |  // 2. Find closest
9   |  closest_resource$ = find_closest_resource resources$...
10  |
11  |  // 3. Get energy
12  |  move_to location@ of closest_resources$
13  |
14  +---end
15
16  instruction find_closest_resource with resources$... means
17   |
18   |  closest_distance# = MAX#
19   |  closest_resource$ = NOTHING$
20   |  my_loc@ = location@ of self
21   |
22   |  repeat with each resource$ in resources$...
23   |
24   |    |  resource_loc@ = location of resource$
25   |    |  distance# = distance resource_loc@ my_loc@
26   |
27   |    |  if distance# is_less_than closest_distance# then
28   |    |    |
29   |    |      |  closest_distance# = distance#
30   |    |      |  closest_resource$ = resource$
31   |    |
32   |    |    +---done
33   |
34   |    +---done
35   |
36   +---gives closest_resource$
```

The main changes to note in this sample are the new data type $ which signifies that our variable is a resource. Resources contain a location and a type (the types of resources or "power-ups" are listed above in section 2). The other key difference in this code is the use of move_to instead of shoot as the robot will automatically collect a resource upon arriving at its location.
Now we'll try to put everything together in the following example which uses the previous two functions for determining the closest enemy and resource in order to execute a simple battle strategy.

```plaintext
[Sample code 6 : Destroy all enemies]

think
1 | if energy_level% of self is_greater_than 10% //if energy is less than 10% then
2 | wait to recharge
3 | ping_radar // Decreases energy
4 | // 1. Get enemy locations
5 | enemies!... = get_enemies
6 | if enemies!... is_not NOTHING!
7 | |
8 | // 2. Find closest
9 | closest_enemy! = NOTHING!
10 | closest_enemy! = find_closest_enemy enemies!...
11 |
12 | closest_resource$ = NOTHING$
13 | resources$... = get_resources
14 |
15 | if energy_level% of self is_less_than 30% then
16 | closest_resource$ = find_closest_resource resources$...
17 | move_to location@ of closest_resource$
18 | else //fight!
19 | dist# = distance location@ of closest_enemy! location@ of self
20 | if dist# < 10
21 | shoot closest_enemy! // Decreases energy
22 | |else
23 | | move_to location@ of closest_enemy!
24 | | +---done
25 | | +---done
26 | | else //There's nobody around to fight
27 | | say I'm bored!!
28 | | +---done
29 | +---done
30 +---end
```

This program utilizes many conditionals in order to choose the best course of actions in the middle of a fight depending on the current circumstances. We see the first conditional on line 2 which determines if the robot has more than 10% of its energy in reserves. If it does not then we decide to sit and wait for it to recharge, this is a simple default to protect against running out of energy in the middle of a fight. A wiser strategy could involve using radar data from previous think iterations in order to chase after close resources or attack a nearby enemy which was recently detected. In order to keep this example simple however, we simply choose to wait for our battery to recharge before we continue operation.

If we do have enough energy in reserves then we ping our radar to get the latest inventory of enemies and resources within the radars range. If there are no enemies within range then we simply state "I'm bored!!". If there are enemies in range then we probably want to keep moving, either by fighting or taking all the resources in the area and make ourselves a difficult target to hit. Once determining that there are enemies within range of the radar, we find the closest enemy using our `find_closest_enemy` function. On line 18 we make our next decision, if our energy is below 30% then we decide that fighting could be a dangerous option as we might run out of energy mid battle, instead we choose to go after resources in hopes of gaining more fighting ability and making ourselves difficult to attack. We use our `find_closest_resources` function and then move to that location. If our energy is greater than 30% then we decide to fight. Rather than wasting our energy firing at enemies which are far away, we check the distance and if its greater than 10 meters then we move towards our enemy at 100% speed, otherwise if we are in fact in range, we fire on our opponent.
Make ROBOT your own

Now that you’ve completed all the practice programs in this tutorial you should have all the tools you need to begin experimenting and creating your very own ROBOT programs. While this tutorial covered all the main aspects of the ROBOT programming language, there are still many operations and library functions which are not discussed here. Remember to utilize both the resource manual and the IDE’s toolbox in order to extend your options when writing ROBOT code and even when times get tough, keep in mind that ROBOT is a game so just relax, have fun and destroy everything!