**Fall 2012**

**Drone Language Manual**

**George Brink gb2280**

**Xiaotong Chen xc2230**

**Shuo Qiu sq2144**

**Xiang Yao xy2191**

**Introduction:**

Drone language is a stack-based imperative language. Designed to be used in a Drone War game.

The stack accepts only integers, booleans, and flags. Integers can be used as arithmetic operands or parameters of the functions. Booleans are subject to stack manipulation operations and as parameter for conditional jump operators. Flags are subject to stack manipulation operations and parameters for special functions which check is the flag is of the expected kind and leave boolean true or false on the stack.

Each word read from the source code is either a comment, integer, boolean call to a user defined function, label, variable, or operator.

**1. Language Syntax**

**1.1 Keywords**

Keywords used by the language are case insensitive (i.e Dup is the same as DUP or dup). The list of known keywords is:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| dup | Drop | dropAll | swap | over | rot | read | store | jump | jumpIf |
| sub | endSub | move | stop | shoot | look | wait | getHealth | random | Mod |
| isFoe | isAlly | isWall | isEnd | and | or | not | if | else | endif |

**1.2 Player defined names**

Unlike keywords (which are case insensitive), names defined by the player are case sensitive.

Those names are used as names for variables, labels, and user defined functions.

**1.3 Comments**

Single line comments, start with a word // and continue to the end of the line. E.g. each of the following lines contains a comment

E.g.

// whole line can be a comment

2 2 + // or comment can start after some compilable words

// any word appeared after first // is still a comment

Multi line comments, start with a word /\* and continue to the first \*/ word. The nested comments are not supported.

E.g.

/\* Inside here is a comment \*/

**1.4 Functions**

**1.4.1 Structure of Function**

User functions are marked with a word "Sub" followed by a function name, any number of commands and ends with "EndSub".

Sub foo

these words a body of a function

EndSub

Sub myAdd

+

EndSub

It is not possible to redefine

**1.4.2 Call of Function**

The call to the user defined function is just its name. E.g. assuming we defined the function ‘myAdd’ as in the previous chapter, then the next two lines will do exactly the same:

2 2 +

2 2 myAdd

**1.4.3 Layer of Function**

Functions cannot have sub-functions. For example, the next example shows an illegal code:

Sub foo 1 2

Sub bar // error

3 4

EndSub

EndSub

**1.5 Label**

**1.5.1 Structure of Label**

Labels start with a letter followed by any number of letters, numbers, and ‘\_’ (underscore) symbols. Labels ended with colon:

this\_is\_label:

this-is/not\_a.label:

123456: // also not a label

Of course, the white-space character split sequences of characters into sequence of words and the next line will be understood as four words and a label with the name 'label':

this is not a label:

**1.5.2 Unconditional and conditional jumps to the label**

Operation "unconditional jump to the label" is marked by adding, "jump" to the name. The next line shows an unconditional jump to the labels defined in the previous example:

this\_is\_label jump

Conditional jump (marked jumpIf) checks the top of the stack first, if there was a true value, then the jump happens, if there was a false value, then jump does not happen and the execution is passed to the next operation after jumpif.

**1.5.3 Local & Global Labels**

Label visibility is restricted to the function. For example:

Sub foo

2

lbl1: 2 +

lbl1 Jump // ok

EndSub

lbl2: lbl1 JumpIf // error

Here, label lbl1 is defined inside a function foo and jump to it is allowed. The label lbl2 is defined in the main program and jump to it is allowed from any where from the main program, but not from the inside of user defined function. Conversely, the conditional jump to lbl1 will fail since the label is defined inside of the function, but the jump is attempted from the main program.

**2. Fundamental Types**

**2.1 Integer**

Integer is word which consists solely from characters 0-9.

123 // one integer

1 2 3 // three integers

These words put the specified integer directly on the stack.

**2.2 Boolean**

Booleans are two words "true" and "false" which represent the logical values and are subject to logical operations and conditional jumps.

**2.3 Flags**

Flags are game specific type. They are produced by the function Look and explain what drone sees. There are four such flags: Foe, Ally, Wall, and End.

**3. Variables**

Variables are words started with a letter and any number of letters or digits that directly followed by keywords "store" or "read". The first one take the top of the stack and stores it into the variable (creating the variable in the process if necessary). The second one reads variable and puts its contents on the stack. E.g.

2 abc store

assign the value in the top of stack to variable abc. in this example, we assign 2 to abc so that we can use abc in the future

abc read

abc read means we get the content of variable abc and push it into the stack. In this example, we push 2 to the stack because we assigned 2 to abc before.

Variables can contain any of the three fundamental types: integer, boolean or flag.

**4. Operators**

Operators are always taking some number of values from the stack and return some values back on the stack:

**4.1 Arithmetic operators**

+ a b -> (a + b)

- a b -> (a - b)

\* a b -> (a \* b)

/ a b -> (a / b)

mod a b -> (a mod b)

^ a b -> (a ^ b)

**4. 2 Logic operators**

and a b -> (a and b)

or a b -> (a or b)

not a -> (not a)

**4. 3 Logic constants**

true -> true

false -> false

**4.4 Conditions**

= a b -> (a = b)

< a b -> (a < b)

> a b -> (a > b)

**4.5 Stack manipulation**

drop a b c -> a b

dropall a b c ->

dup a b c -> a b c c

swap a b c -> a c b

over a b c -> a b c b

rot a b c -> b c a

**4.6 Read & Store**

name store

Store value into variable "name", create the variable if necessary. Always read the first on the stack and value it to “name”.

name read

Read value from variable "name". Die if such variable does not exist.

**5. Game specific functions**

**5.1 move** direction **->**

Start moving in the specified direction

**5.2 stop ->**

Stop moving

**5.3 shoot** distance direction **->** bool

Shoot in the specified direction. Projectile will explode after traveling the specified distance, and returns boolean value:

true -> shooting was successful and projectile is on its way

false -> cannon did not have enough time to cool-down

**5.4 look** direction -> END dist-1 dir-1 type-1 [... dist-n dir-n type-n ]

Look for other drones and walls in the specified direction. Returns one or more triplets (distance, direction, and type) which represent distance to the object, the exact direction to the object and type of the object. Type of the object is a flag from the set: FOE, ALLY, WALL. After the last triplets there would be a special flag END, which represents end of the look’s output.

**5.5 isFoe** flag -> bool

Checks is the top of the stack contains a flag FOE and returns corresponding boolean value.

**5.6 isAlly** flag -> bool

Checks is the top of the stack contains a flag ALLY and returns corresponding boolean value.

**5.7 isWall** flag -> bool

Checks is the top of the stack contains a flag WALL and returns corresponding boolean value.

**5.8 isEnd** flag -> bool

Checks is the top of the stack contains a flag END and returns corresponding boolean value.

**5.9 wait** ticks ->

Be idle (do nothing) for specified number of ticks

**5.10 getHealth** -> integer

Put current drone's health on the stack

**5.11 random** a b -> integer

Make a random integer in the range [a,b] (inclusive) and return it.

**6. Pseudo-commands**

All operators and game-specific commands described in sections 4 and 5 take exactly are executed directly by the game engine and take one tick to perform. The next set of commands added for convenience. They are compiled by the translator into several simple operators and can take any number of additional ticks to complete.

**6.1 Conditions**

Conditional branching is done by the means IF/ELSE/ENDIF. The stack should contain a Boolean value before the IF. If this value is true, then the set of command which follows the IF would be executed. If the value is false, then control jumps to the set of commands after the keyword ELSE, or to the command which follows ENDIF, if the ELSE keyword is omitted. Nested IF branching is allowed. For example, shoot if the top of the stack contains the description of the enemy drone

isFoe if shoot endif

This code will be transformed by compiler into:

isFoe not endif\_label jumpIf shoot endif\_label:

**6.2 Loops**

**6.2.1 Endless loop**

The endless loop is the most simple one, it is defined by keywords BEGIN and AGAIN:

begin 100 500 random 0 360 random shoot again

This will make the drone to shoot endlessly to a random distance in a random direction. This code is converted into a simple:

label: 100 500 random 0 360 random shoot label jump

**6.2.2 Conditional loop**

Conditional loops are defined by the same BEGIN and AGAIN keywords. Addition of the WHILE keyword allows to leave the endless loop if top of the stack is false when execution reach the WHILE keyword. For example, the cleanup after the LOOK command can be like this:

begin isEnd while drop drop again

This is code will be compiled into

label\_start: isEnd jumpIf label\_end drop drop label\_start jump label\_end:

The WHILE keyword can appear anywhere inside the BEGIN-AGAIN block, this allows to create loops with post-conditions or even with conditions in the middle of the block:

begin dup isfoe shoot endif isend while drop drop again

Both types of loops can be nested.