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 Whitespace & Word

Language syntax consists of "words" separated by whitespace. A word is a sequence of arbitrary characters (except whitespace).
E.g., each of the following lines contains exactly one word:

```
word
!@#$%^&*()
1234567890
5!a
```

1.2 Capitalization

Language is case insensitive.
E.g. each of the following words is exactly the same:

```
Word
wOrd
WORD
WoRd
```

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 // 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 and ends with "esub".
E.g.

```
sub foo
        these words a body of a function
esub
sub add
        +
esub
```

1.4.2 Call of Function

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

```
2 2 +
2 2 add
```

1.4.3 Layer of Function

Functions should have simple layer, that is, functions cannot have sub-functions. For example, the next example shows an illegal code:

```
sub foo 1 2
        sub bar   // error
                3 4
          esub
esub
```

1.5 Label

1.5.1 Structure of Label

Labels are any set of characters ended with colon:

```
this_is_label:
this-is/also.label:
123456:
```

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 three lines are jumps to the labels defined in the previous example:

```
this_is_label jump
this-is/also.label jump
123456 jump
```

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

1.5.3 Local & Global of Label

Labels visibility are restricted to the function. For example:

```
sub foo
        2
        lbl1: 2 +
        lbl1 jump
esub
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.
E.g.

```
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.
E.g.

```
true
false
```

2.4 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.

3. Operators

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

3.1 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)

3. 2 Logic operators

and	a b -> (a and b)
or	a b -> (a or b)
not	a -> (not a)

3. 3 Logic constants

true	-> true
false	-> false

3.4 Conditions

=	a b -> (a = b)
<	a b -> (a < b)
>	a b -> (a > b)

3.5 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

3.6 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.

4. Game specific functions

4.1 move direction(integer) -> void

Start moving in the specified direction

4.2 stop -> void

Stop moving

4.3 shoot distance(integer) direction(integer) -> 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

4.4 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 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.

4.5 isFoe flag -> bool

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

4.6 isAlly flag -> bool

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

4.7 isWall flag -> bool

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

4.8 isEnd flag -> bool

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

4.9 wait milliseconds(integer) -> void

Be idle (do nothing) for specified number of milliseconds.

4.10 getHealth -> integer

Put current drone's health on the stack

4.11 random a(integer) b(integer) -> integer

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