The Drone War

Project Proposal

COMS W4115 Programming Languages and Translators
Columbia University, Fall 2012
Professor: Stephen A. Edwards
Students:
George Brink gb2280
Xiaotong Chen xc2230
Shuo Qiu sq2144
Xiang Yao xy2191

Motivation and Introduction:

We want to implement a language that first of all can be interesting for regular users, so we decided to design a game which will use our language.

In the Drone War, which belongs to the "programming game" genre, the player has no direct influence on the course of the game. Instead, a player writes a program which acts as an AI for the "drone". Several drones (each with its own AI program) are randomly dropped into the arena. Drones are fighting with each other until only one is left or the time limit for the battle is exceeded.

The language for the drones' AI has some requirements which are based in the nature of the game:

1) It has to be simple enough to be understood by players who is not very experienced in the art of programming. Therefore the language requires to have as few operators and concepts as possible.

2) Since all drones are acting simultaneously, there is a good chance that smart but inefficiently written AI will lose to the not so smart but faster AI. Therefore, the "machine" language for drones requires a high level of predictability on how long each operation will take. The ideal solution for this is a global rule: one tick per operation.

We decided that to satisfy both these two requirements the best solution would be a stack based language. The stack paradigm allows us to stick with the postfix notation, and we can clearly state that each operation takes just one tick of drone's CPU.

In the language we designed, will not use predefined variables, but will create them on the first attempt to store a value. Variables and the stack accept values of different types (integers, boolean, or flags).

Introduction to Drone War:

At the start of the battle, drones are put on the Arena at random. Arena is a square enclosed by impenetrable walls. Drone which hits the wall receive some damage. Size of the arena is 1000 * 1000 units.

Drone represents a land vehicle with a freely turning cannon (meaning a drone can move in one direction while shooting in another). All drones are equal (safe for the AIs). Drone have 100 health points at the start of the battle. Once drone loses all its HP it disappears from the arena.

If drone's AI attempt to do an illegal operation (for example divide by zero or use a logic operation on the value of integer type) – drone should freeze and do nothing until the battle is over.

Drone can move around the arena by issuing command: *move* with one parameter *direction*. Once the command is issued, the drone starts moving in the desired direction until next *move* command changes it or the *stop* command cancel the movement. Drone does not have "mass" so there is no need to worry about inertia. If drone hits the wall of the arena or another drone, it loses 10 HP as a result of the hit. The movement speed is set to 100 points per second (it will take ten seconds for the drone to cross the arena from one wall to another).

Drone can attack enemies by issuing a command: *shoot* with two parameters *direction* and *distance*. This command will send a projectile in specified direction. If the projectile hit something (another drone or wall) it explodes. If projectile did not hit anything – it explodes after flying the specified distance. Explosion damages drones, the closer to the epicenter the bigger the damage.

Explosion has a radius of 50 points, meaning that if a drone was hit directly it receives 50 points of damage. If distance to the epicenter was 1 point, drone receives 49 points of damage. Distance of 50 points or more is completely safe. A drone can be damaged by its own projectile if it blows up close enough. Projectile's speed is 500 points per second. Projectile cannot travel for more than 500 points (half of the arena).

Drone cannot issue a *shoot* command more than 5 times per second. If it attempt to do so – the *shoot* will return FALSE. If shooting was successful – TRUE. NB: this return code does not tell was the target hit or not!

Drone can see other drones and walls of the arena by issuing a command: *look* with one parameter *direction*. *Look* has an "angle of vision" with the side angle of 10 degrees. This means, the drone sees not just objects on the straight line but in the area of a triangle. The distance to the wall is calculated by the exact direction of the *look*.

The *look* command returns a list of integers and flags:

```
END distance_1 direction_1 type_1 [... distance_N direction_N type_N]
```

Where each tuple (distance, direction, and type) describes one of the object the drone saw. The 'type' flags can be FOE, ALLY, or WALL. The special flag END indicates that there are no more tuples.

Each command of the AI takes exactly one tick to process. So the more advanced AI can be potentially slower than simpler ones.

Drone Language

Drone language is a stack-based imperative language.

Data types

The stack accepts three data types: 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.

Language syntax

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

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

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

```
Word
wOrd
WORD
WoRd
```

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

Comments

Comments are single line only, they start with a word // and continue to the end of the line. E.g. each of the following lines contains a comment:

```
// whole line can be a comment
2 2  + // or comment can start after some compilable words
// any word appeared after // is still a comment
```

Integers

Integer is a 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.

Booleans

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

User functions

User functions are marked with a word which starts with a colon, and ends with a single colon.

```
:foo these words a body of a function :
:add + :
```

The first example defines a function named "foo" with some arbitrary words and the second example defines an alias "add" for a simple arithmetic operation.

The call to the previously defined function is just its name. E.g. the nexttwo lines will do exactly the same:

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

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

```
:foo 1 2 :bar 3 4 : :
```

Labels

Labels are any set of characters ended with colon:

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

Of course, the wight-space character splits any sequence 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:
```

Labels visibility are restricted to the function. For example:

```
:foo 2 lbl1: 2 + lbl1> : lbl2: lbl1?
```

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.

Jumps

Operation "unconditional jump to the label" is marked with adding character ">" to the name. The next three lines are jumps to the labels defined in the label examples:

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

The conditional jump is marked by adding character '?' to the name. Conditional jump happens if and only if the top of the stack contains boolean "true" value. If top of the stack is not a boolean value, then drone freezes.

Variables

Variables are words which are directly preceded by characters ">" or "<". 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. For example, if you need to store top value in the variable and leave it on the stack for the future use, you can do this:

```
>abc <abc
```

Operators

Operators are always take some number of values from the stack and return some values back

on the stack

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)
```

Logic operators:

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

Logic constants

```
true            -> true
false           -> false
```

Conditions:

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

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
```

Prefixes:

```
>name        a ->
```

Store value into variable "name", create the variable if necessary.

```
<name           -> a
```

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

```
:name           ->
```

Start definition of function "name". Function definition ends with single character ':'.

Suffixes:

```
name:           ->
```

Define a label.

```
name>           ->
```

Unconditional jump to the label.

```
name?        a ->
```

Jump to the specified label if top of the stack contains boolean true. Continue execution of the next word if there was false. Die if top of the stack was not a boolean value.

Game specific operators:

```
move   direction ->
```

Start moving in the specified direction

```
stop            ->
```
Stop moving

```
shoot  distance direction -> bool
```
Shoot in the specified direction. Projectile will explode after traveling the specified distance. Returns boolean value:

true = shooting was successful and projectile is on its way
false = cannon did not have enough time to cool-down

```
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. Under the last triplets there would be a special flag END.

```
isFoe  flag -> bool
```
Checks is the top of the stack contains a flag FOE and returns corresponding boolean value.

```
isAlly  flag -> bool
```
Checks is the top of the stack contains a flag ALLY and returns corresponding boolean value.

```
isWall  flag -> bool
```
Checks is the top of the stack contains a flag WALL and returns corresponding boolean value.

```
isEnd  flag -> bool
```
Checks is the top of the stack contains a flag END and returns corresponding boolean value.

```
wait   ticks ->
```
Do nothing for specified number of ticks.

```
getHealth    -> health
```
Put current drone's health on the stack

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

Examples of drones:

Rabbit.dt

```
// The extremely harmless drone.
// It sits on one place and checks its health
// If damage detected - run somewhere for 0.1 seconds in hope to
// leave the the zone of danger. Then stop and wait until
// it again receive some damage.

100 >health               // set variable health to 100

main_loop:
      10 wait             // wait for 0.1 of the second

      // read the stored value of health
      // read the current health and compare it with the old value
      <health getHealth =
     // inverse boolean value, and jump to the I_was_hurt if the result is true
      not I_was_hurt?

      // repeat indefinitely
      main_loop>

// sub program, what to do if drone received some damage
I_was_hurt:
      0 359 random        // get a random value in the range 1-360
      move                // move in the random direction
      10 wait             // wait for 0.1 seconds
      stop                // stop

      main_loop>          // and go back to the main loop
```

Turret.dt

```
// The “stand and shoot” drone.

// make a variable named 'direction' and store value 0 in it
0 >direction

main_loop:
      // clean the stack (it can overflow by multiple calls to look)
      dropall
      // load value from the variable and do the look operation
      <direction look

      // Look dumps a list of triplets distance-direction-type to objects found
      // in the direction of look, we are interested only in the closest one.
      isFoe
      // if the object on top of the stack was a foe, jump to shootIt
      shootIt?

      // else increment the value of the 'direction' variable by 10
      <direction 10 + >direction
      // go to the beginning
      main_loop>
```

```
// this sub program will work if drone saw an enemy
shootIt:
      // store the exact direction to the object for future ease of targeting
      dup >direction
      // shoot...
      shoot
      // wait for gun to cool down...
      10 wait
      // and go to the main loop
      main_loop>
```

Berserk.dt

```
// This drone is very aggressive. It looks for any other drone, regardless of
// is it friend or foe, runs toward it and shoot.

0 >direction

main_loop:
      <direction look

check_what_I_saw:
      isDrone sees_a_drone?
      isEnd main_loop?
       // if that was not a drone or END, then it had to be a WALL
      // drop direction and distance to the wall and repeat the deciphering loop
      drop2
      check_what_I_saw>

sees_a_drone:
      drop              // it should have an ALLY/FOE flag after isDrone
      dup >direction    // store the direction to the drone
      move              // start moving toward the target
      <direction shoot  // and shoot in the same direction

      // after charged to the nearest drone, we still have to cleanup data for
      // other objects seen by look.
look_cleanup:
      isEnd main_loop?
      drop2
      look_cleanup>


// user function, (convenient but inefficient)
// drop two values from the stack
:drop2 drop drop :


// User function
// Push TRUE to the stack if top value is a flag which represents a drone
// Keep the flag itself.
:isDrone
      dup isAlly l1?
      dup isFoe
      l2>
l1:
      true
l2:   :  // return from the user function
```