CMPT 370
RobotSport370 Language

Christopher Dutchyn

September 19, 2015

The RoboSport370 control language (RS370CL) is modelled on Forth, but simplified and customized for our specific purposes.

Forth is post-fix, stack-based language, where integers, booleans, and strings are

- pushed by callers as arguments to procedures (called *words*),
- popped off by the word and operated on
- pushed back on as results from the word.

In addition to the core procedures supporting our subset of Forth, there is a small library of words which interface the robot to the simulator. A program defines a set of your own words, including an `init` word to allow a robot to initialize itself, and a `turn` word which the simulator calls whenever the robot's turn arises. Examples of a sequence of words that computes useful values include:

`1234 567 +` — computes 1801
`50 random` — computes a random number between 0 and 49

Literal Words

There are three types of literal words, corresponding to pushing simple values:

1. Integers: written as decimal numbers (with or without a leading sign (`+` or `-`). Examples are:

   ```
   100
   -5
   ```

2. Strings: written as sequences of characters (including whitespace) initiated by the special token `."` and ended by the next `"`. These will only be used for debugging, logging, and inter-robot communication. The simulator does not require them, but the robots will. Examples are

```
." This is a string with 7 spaces."
."ThisOneHas37Characters-NoneAreSpaces!"
```

Note that the string value itself does not include the surrounding double-quotes, and there are no escape characters.

3. Booleans: written as the special tokens, `true` and `false`.

Defining Words

Every programming language needs a way to extend itself. Forth does this by starting a new definition with a colon (`:`), then the name of the new word, then a series of Forth words which form the body of the word, and a terminating semicolon (`;`). When the `move` word is called by the simulator, each of the words in the body are executed, one at a time, in order. If the word to be executed is a literal, it is pushed onto the stack. If the word to be executed is a primitive, system-defined word, then the operation is performed. If the word is another user-defined word, then the body of that second word is executed and control returns to the next word in the original definition. For example,

`: double dup + ;` — double an integer
`: triple dup double + ;` — triple an integer
`: quadruple  double double ;` — quadruple an integer

`3 double` — computes 6 (and leaves it on the top of the stack)

Comments

No programming language and no program is complete without some form of comments. Comments are delimited by parentheses:

```
1234 ( this is a comment ) 456 +
```

One particularly useful, and conventional form of comment is a *stack* comment, which tells a programmer what effect the word has on the stack. Its typical format is a collection of letters encoding the expected order of arguments on the stack, all of which will be popped off,

`i` : integer

`s` : string

`b` : boolean

`v` : any type of value

`p` : a variable (aka pointer, see below)

followed by a double-dash (`--`) and then a collection of letters encoding the expected order of results left on the stack. The top of the stack is always closest to the double-dash, and we use numbers to distinguish values where useful. Examples are:

`: double ( i -- i ) 2 * ;` — defines a word that doubles an integer on the stack

`: stutter ( s i -- ) ... ;` — defines a word that repeats a string, returning nothing

Note that the integer is at the top of the stack, because the string is pushed first:

```
."um" 3 stutter
```

and that in the following example, the quotient is at the top and a boolean is below it:

`: div2 ( i -- i b ) 2 /mod swap 0 <> swap ;` — halves a number, returns quotient and remainder flag

so that the following word is a complicated way of doing nothing:

```
: id ( i -- i ) div2 double swap if 1 else 0 then + ;
```

If this example is confusing, it will make more sense after we see some more standard Forth words. The maxim is, *push items in left-to-right order as before --, expect to pop items in left-to-right order as after --.*

Forth Words

We include a number of standard Forth words for stack-manipulation routines, arithmetic routines, and operations.

Stack Words

These manipulate values on the stack:

- `drop ( v -- )` —remove the value at the top of the stack
- `dup ( v -- v v )` —duplicate the value at the top of the stack
- `swap ( v2 v1 -- v2 v1 )` —swap the two values at the top of the stack
- `rot ( v3 v2 v1 -- v3 v1 v2 )` —rotate the top three elements of the stack

Arithmetic Words

Arithmetic is necessary for computing distances, directions, etc.

- `+ ( i i -- i)` —add the two integers, pushing their sum on the stack
- `- ( i2 i1 -- i )` —subtract the top integer from the next, pushing their difference on the stack Note that `4 3 -` means subtract 3 from 4, giving 1.
- `* ( i i -- i )` —multiply the two top integers, pushing their product on the stack
- `/mod ( iv ie -- iq ir)` —divide the top integer into the next, pushing the remainder and quotient (in that order) Note that `47 5 /mod` means divide 47 by 5, leaving 9 at the top of the stack, and 2 next.

Comparisons

Integers can be compared, yielding a boolean that indicates the result:

- `< ( i2 i1 -- b )` —`i2` is less than `i1`
- `<= ( i2 i1 -- b )` —`i2` is not more than `i1`
- `= ( i2 i1 -- b )` —`i2` equals `i1`
- `<> ( i2 i1 -- b )` —`i2` is different from `i1`
- `=> ( i2 i1 -- b )` —`i2` is at least `i1`
- `> ( i2 i1 -- b )` —`i2` is more than `i1`

Logic and Control

Booleans can be combined and modified

- `and ( b b -- b )` —`false` if either boolean is `false`, `true` otherwise
- `or ( b b -- b )` —`true` if either boolean is `true`, `false` otherwise
- `invert ( b -- b )` —invert the given boolean

for use in conditional expressions

`...`*`test`*`... if ...`*`true-branch`*`... else ...`*`false-branch`*`... then`

such as

```
1 1 = if ."equal" else ."unequal" then
```

will push the string `equal` onto the stack, and

```
1 1 <> if ."equal" else ."unequal" then
```

will push the string `unequal` onto the stack. The essential structure is

- the *test* expression (that yields a boolean) comes before `if`,
- the *true-branch* (the words to execute if the test is true) come before `else`,
- and *false-branch* (the words to execute if the test is false) come before `then`

and the words to execute after the conditional come after `then`. Note that conditionals can be nested, as this example shows:

```
: spotter ( i -- ) dup mesg? if dup
                                recv! ."SHOOT" =
                                if dup
                                   recv! recv! shoot!
                                else drop ."not shooting"
                                then
                             else drop ."no message"
                             then ;
```

will define a word that will take a number and check if a message has arrived from that teammate. If one has, it will read a string from that teammate; and if the string is `SHOOT`, then it will read two integers from that teammate (presumably a range and angle) and shoot at that spot. Otherwise, it needs to clean up the stack.

Loops

The language also includes counted and guarded loops, as well as a way to break out of a loop. First, the counted loop:

...end... *...start...* `do` *...body...* `loop`

which takes two integers (computed by the *end* and *start* expressions) and repeats the *body* words for *start*, *start+1*, ..., *end-1*, *end*. If *end* is less than or equal to *start*, then *body* will only be executed once. The current value for the repetition (i.e. *start*, then *start+1*, ...) are available as `I` within the loop. The `leave` word immediately exits the current loop. Note that loops can nest, so `leave` will only exit the innermost loop. For example, this simple robot program simply checks each of its immediate neighbours (in clockwise order)and shoots the first

```
: turn ( -- ) 5 0 do I 1 empty? if ."no one there"
                                   else I 1 shoot! leave
                                   then
                      loop ;
```

If you want to count down (i.e. counter-clockwise order), simply perform the subtraction against *end*:

```
: turn ( -- ) 5 0 do 5 I - 1 empty? if ."no one there"
                                    else 5 I - 1 shoot! leave
                                    then
                  loop ;
```

If you want to count by more than one, simply perform the multiplication. This robot simply shoots the spaces at range 2 which are in direct line-of-sight:

```
: turn ( -- ) 5 0 do I 2 * 2 empty? if ."no one there"
                                    else I 2 * 2 shoot! leave
                                    then
                  loop ;
```

Second, the guarded loop:

`begin` *...body...* *...finished...* `until`

which repeats *body* as long as the *finished* computes to `false`. We could write the first clockwise shooter above as

```
: turn ( -- ) 0 begin dup 1 swap empty? if ."no one there"
                                        else dup 1 shoot! leave
                                        then
                      1 + dup 5 >
                until drop ;
```

Variables

We allow the robot programmers to declare variables (storage locations), and to store and retrieve values from those locations. These operations are

- `variable` *name* `; ( -- )` —declares a new variable, initially zero
- `? ( p -- v )` —takes a variable and returns the value it's storing
- `! ( v p -- )` —stores a new value into a pointer

Variables can contain strings, but they do start off containing an integer. Note that constants are not necessary, because simple defined words can serve the same purpose:

```
: maxRange 3 ;
: hexesPerRange 6 ;
```

A smarter clockwise-shooter, that picks up from where it left off, might be written as

```
variable lastShot ;
: init 0 lastShot ! ;
: turn ( -- ) 0 begin dup lastShot ? + 1 6 /mod drop
                          empty? if ."no one there"
                                 else dup lastShot !
                                      dup 1 shoot! leave
                                 then
                       1 + dup 5 >
                until drop ;
```

Miscellanea

Our Forth subset includes a couple of utility functions:

- `.  ( v -- )` —prints the current value out, using the same syntax as they would be input with
- `random ( i -- )` —generates a random integer between 0 and `i` inclusive

A simple random shooter (that won't kill itself) might be

```
: maxRange 3 ;
: hexesPerRange 6 ;
: turn ( -- ) maxRange 1 - random 1 +
              dup hexesPerRange * 1 - random
              swap shoot! ;
```

Robot Library

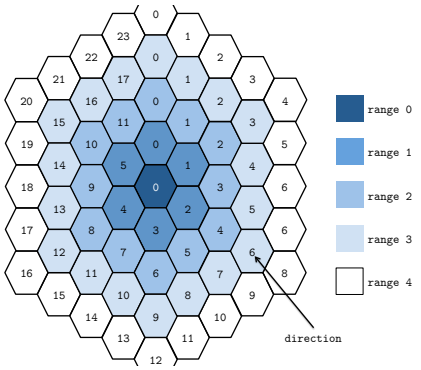
The robot programmer relies on a library of words to access robot actions, robot status, and battle state.

Status

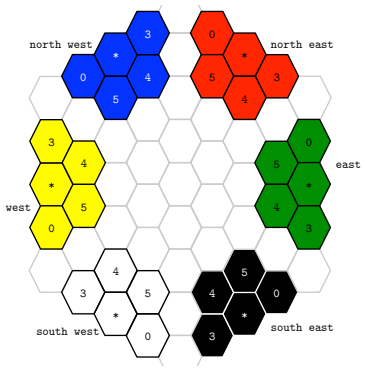
Robots can query their own state:

- `health ( -- i )` —returns the robot's current health (1–3)
- `movesLeft ( -- i )` —returns the robot's range of movement (0–3), movement regenerates each turn and is consumed as moves are made
- `firepower ( -- i )` —returns the robot's firepower (1–3), which never changes
- `team ( -- i )` —returns the robot's current team number assigned by the simulator: all robots on one team have the same team number
- `member ( -- i )` —returns the robot's member number (0–3) assigned by the simulator when the robot enters the board

Figure 1: Directions and Ranges for EAST team



Directions for each team



Actions

Robots can shoot, move, and scan. Each of these involves directions and ranges, described by left-half of the following diagram: At longer ranges, the concentric number continues. Note that the zero-direction on the whole board differs for each team: it is perpendicular to the side the robots enter on. The image is for a robot entering from the east; for the other sides, see the right-half of the diagram.

The actual words for controlling a robot are:

- `shoot!  ( id ir -- )` —fires the robot's weapon at the space at range `ir` and direction `id`; any robots in that space (including the shooter if range is zero) reduce their health by an amount equal to the shooter's `firepower`. Robots may only shoot once per turn.
- `move!  ( id ir -- )` —moves the robot to the space at range `ir` direction `id`, provided they have enough `movesLeft`; otherwise they do not move at all. Crashing into the wall also prevents the move from occurring.
- `scan!  ( -- i )` —scans for the nearest robots, and reports how many targets visible, up to four. Distance ties are broken randomly.
- `identify!  ( i -- it ir id ih )` —identifies the given target, giving its team number (`it`), range (`ir`), direction (`id`), and health (`ih`).

Using these words, a simple sniper might be:

```
variable canShoot ;
: canShoot? canShoot ? ;
: shoot!! canShoot false ! shoot!
: maxRange 3 ;
: inRange? dup maxRange <= ;
: enemy? team <> ;
```

```
: turn ( -- ) canShoot true !
              scan!
              0 do enemy? if inRange? if canShoot? shoot!!
                                      else ."out of range" pop pop
                                      then
                           else ."avoid friendly fire" pop pop
                           then pop
            loop ;
```

Sophisticated robots might build a map by interrogating the contents of each hex, using

- `hex ( id ir -- )` —returns the population of the given hex

which could be used to derive the `empty?` word we used in examples above:

```
: empty? ( id ir -- ) hex 0 = ;
```

Mailboxes

To allow coordinated action among robots on a team, each robot is equipped with mailboxes, one for each team member, to receive messages from other robots. Each receiving mailbox (one for each other team member) can store up to six (non-pointer: i.e. string, integer, or boolean) values in the order they're sent from the other robot. In this way, a robot might designate a target, or report status to the others. For simplicity, a robot can send and receive messages to itself: this might be useful for preserving information across `turn`s. Mailboxes are facilitated by five words:

- `send!  ( i v -- b )` —send a value `v` to team-member `i`; returns a boolean indicating success or failure. Failure can occur if the robot is dead (`health` = 0), or its mailbox is full.
- `mesg?  ( i -- b )` —indicates whether the robot has a waiting message from team-member `i`
- `recv!  ( i -- v )` —pushes the next message value onto the stack.

A robot attempting to `recv!` when no message is ready will automatically end its `turn` or `init` word.

Tournaments

A tournament can be held between two teams of robots, three teams of robots, or six teams of robots, on a hexagonal grid with a side of five, seven, nine, or eleven hexes. In a duel, the robots enter on opposite edges of the hex grid; with three competing teams, they enter on non-adjacent edges (i.e. there is an empty edge between), and for six teams, each team starts on one of the edges.

All robots on a team enter in the centre hex of their side, and execute their `init` word from that position. The zero-direction is perpendicular to the side the team enters on.

Robot Interface

The simulator needs to call the `init` word for each robot as the teams are placed in their starting square. The `init` word can perform any action, except `shoot!`: robots can emerge from their entrance hex immediately. Then, once all robots are initialized, the simulator goes round-robin through the robots, executing their `turn` word: robots can perform any number of actions during their turn, except they may only `move!` up to their initial `movesLeft` range and may only `shoot!` once per turn. The stack comments for these two interface words are:

```
: init ( -- ) ... ;
: turn ( -- ) ... ;
```

Team Composition

A team consists of four robots: each robot begins with

- `movesLeft` of 3,
- `firepower` of 1, and
- `health` of 1.

But each robot has three resource points it can expend to improve these scores:

- each additional `health` point expends one resource point, and
- each additional `firepower` point expends one resource point and reduces `movesLeft` by one point.

Simulators and test-benches (for debugging robot programs) may wish to accept printed information (via `.`) and display it in a console log.

Version History

- **09/12/2015: initial 2015T1 version**
- **09/16/2015: renamed `damage` to `health`**