NetLogo 5 Tasks – closures#1

intro

This is another short report relating to tasks in NetLogo 5. This report investigates using tasks to form closures that wrap local variables. This & other reports will go into our repository of Netlogo resources – see www.agent-domain.org.

example

In this example we are looking at using closures to define a stack.

We use the "task" primitive to define 4 closures as follows where the stack itself is held in the variable called "data"...

name	purpose	definition
push	put a new item on the stack	`task [ set data (fput ? data) ]`
peek	return item on the top of the stack	`task [ (first data) ]`
skip	remove item on the top of the stack	`task [ set data (but-first data) ]`
dump	return the stack contents	`task [ data ]`

important points about the way we have organised this...
(i) the 4 closures are stored in a table;
(ii) different versions of the table (representing different stacks) are held in global variables;
(iii) the variable "data" is a local variable.

code for making stacks

```
extensions [table]

globals [*a* *b*]

to-report make-stack
  let data []
  let tab table:make
  table:put tab "push" (task [ set data (fput ? data) ])
  table:put tab "peek" (task [ (first data) ])
  table:put tab "skip" (task [ set data (but-first data) ])
  table:put tab "dump" (task [ data ])
  report tab
end
```

This allows us to manufacture different stacks since each call to *make-stack* uses a different edition of the local variable *data*, see testing below.

testing

```
observer> set *a* make-stack
observer> (run (table:get *a* "push") "A1")
observer> (run (table:get *a* "push") "A2")
observer> show (runresult (table:get *a* "dump"))
observer: ["A2" "A1"]

observer> set *b* make-stack
observer> (run (table:get *b* "push") "B1")
observer> (run (table:get *b* "push") "B2")
observer> show (runresult (table:get *a* "dump"))
observer: ["A2" "A1"]
observer> show (runresult (table:get *b* "dump"))
observer: ["B2" "B1"]

observer> show (runresult (table:get *b* "peek"))
observer: "B2"
observer> show (runresult (table:get *a* "peek"))
observer: "A2"

observer> run (table:get *b* "skip")
observer> show (runresult (table:get *a* "peek"))
observer: "A2"
observer> show (runresult (table:get *b* "peek"))
observer: "B1"
```

comments & issues

The normal way to define a stack is with operations for "push" & "pop". *Push* works as it does above, *pop* removes the top item from the stack and returns it. As a standard NetLogo procedure working on a global list called "stack-data" (which holds the stack contents) *pop* could be implemented something like...

```
to-report pop
  let rtn-val (first stack-data)
  set stack-data (but-first stack-data)
  report rtn-val
end
```

Unfortunately we do not know how to achieve this kind of functionality with the task primitive (as yet).

We cannot use a "*report*" statement inside a task (because report operates on the dynamic context) but neither can we leave a value hanging which would then be returned. So the following is not legal...

```
table:put tab "pop" (task [ let x (first data)
                            set data (but-first data)
                            x
                          ])
```

We do not know of a standard NetLogo form which links/compounds statements, this would allow us to get around the problem. What we want is something like a *Block* statement or the equivalent of Lisp's *prog1* / *progn*).

towards prog1 & progn

Lisp's prog1 and progn forms execute a series of statements returning the value of one of those statements (prog1 returns the value of the first, progn returns the value of the last).

We can make some progress towards the lisp equivalent prog1 & progn by specifying sequences of tasks as long as we can live without dependencies between the tasks in a sequence. What this means in practice is that we cannot declare & use a local variable across a sequence of tasks.

Here is our first attempt at prog1, it runs 2 tasks in sequence, returning then returns value of the first.

prog1 – first attempt

```
to-report prog1 [t1 t2]
  let result (runresult t1)
  ifelse (is-command-task? t2)
  [ run t2 ]
  [ let dummy (runresult t2) ]
  report result
end
```

We use this to specify a new version of make-stack which defines *push* and *pop*...

make-stack – new version

```
to-report make-stack2
  let data []
  let tab table:make
  table:put tab "push" (task [ set data (fput ? data) ])
  table:put tab "pop"  (task [ prog1 (task [ (first data) ])
                                     (task [ set data (but-first data) ])
                                     ])
  table:put tab "dump" (task [ data ])
  report tab
end
```

You will see from the testing below that this works ok but the specification of nested tasks in the code for make-stack2 looks a little clumsy & over-nested.

testing

```
observer> set *c* make-stack2
observer> (run (table:get *c* "push") "C1")
observer> (run (table:get *c* "push") "C2")
observer> show (runresult (table:get *c* "dump"))
observer: ["C2" "C1"]
observer> show (runresult (table:get *c* "pop"))
observer: "C2"
observer> show (runresult (table:get *c* "dump"))
observer: ["C1"]
observer> show (runresult (table:get *c* "pop"))
observer: "C1"
```

prog1 – second attempt

This version allows multiple tasks to be specified (the first version was limited to two), these are passed to prog1 in a list. Unfortunately, though this makes *prog1* more general purpose, it also makes the specification of *pop* even more nested than before.

Note we have now also added a definition of progn...

```
to-report prog1 [tasks]
  let result (runresult (first tasks))
  foreach (but-first tasks)
  [ ifelse (is-command-task? ?)
    [ run ? ]
    [ let dummy (runresult ?) ]
  ]
  report result
end

to-report progn [tasks]
  foreach (but-last tasks)
  [ ifelse (is-command-task? ?)
    [ run ? ]
    [ let dummy (runresult ?) ]
  ]
  report (runresult (last tasks))
end
```

another make-stack

```
to-report make-stack2
  let data []
  let tab table:make
  table:put tab "push" (task [ set data (fput ? data) ])
  table:put tab "pop"  (task [ prog1
                               (list (task [ (first data) ])
                                     (task [ set data (but-first data) ])
                                     )])
  table:put tab "dump" (task [ data ])
  report tab
end
```

using variables

Now we consider how to pass variables across tasks. Since tasks are closures (& therefore wrappers around their local environments) it is not possible to declare a local variable in one task then use it in another. It *is* possible to declare a local variable outside of a task then use it inside the task (we have been doing this already). We test this idea below.

set up code

```
to set-T1&T2             ;; assumes T1 & T2 are globals
  let x 0
  set T1 (list (task [ (set x 1) ])
               (task [ (set x (x + 1)) ])
               (task [ x ])
               )
  set T2 (task [ progn (list (task [ (set x "fruit") ])
                             (task [ (set x (word x "-bat")) ])
                             (task [ x ])
                             )])
end
```

testing

```
observer> set-T1&T2
observer> show (runresult (task [progn T1]))
observer: 2
observer> show (runresult T2)
observer: "fruit-bat"
```

Note that the restrictions we originally faced (in building the stack.pop mechanism) seems to be that reporter-tasks cannot have opening statements which are not reporters themselves. This restriction does not affect command-tasks. So command-tasks can contain blocks of statements. This means that the following works ok...

```
to set-T3
  let x 0
  set T3 (task [ progn (list (task [ (set x "fruit")
                                     (set x (word x "-bat")) ])
                             (task [ x ])
                             )])
end


observer> set-T3
observer> show (runresult T3)
observer: "fruit-bat"
```

Which means that we can rewrite make-stack again like this...

make-stack version 3

```
to-report make-stack3
  let data []
  let tab table:make
  table:put tab "push" (task [ set data (fput ? data) ])

  let x []
  table:put tab "pop"
            (task [ progn (list (task [ set x (first data)
                                        set data (but-first data) ])
                                (task [x])  )])
  table:put tab "dump" (task [ data ])
  report tab
end
```

version 3 tests

```
observer> set *d* make-stack3
observer> (run (table:get *d* "push") "D1")
observer> (run (table:get *d* "push") "D2")
observer> (run (table:get *d* "push") "D3")
observer> show (runresult (table:get *d* "dump"))
observer: ["D3" "D2" "D1"]
observer> show (runresult (table:get *d* "pop"))
observer: "D3"
observer> show (runresult (table:get *d* "pop"))
observer: "D2"
observer> show (runresult (table:get *d* "dump"))
observer: ["D1"]
```

final notes

This is a work in progress. I think there are probably other solutions & some of these may be neater or more flexible than those above. If you have any suggestions please get in touch [S].