CSC302 2011S Programming Languages

Laboratory: Io (3)

Summary: We continue our exploration of the Io programming language.

Prerequisites: The first Io lab and the second Io lab. Sections 3.4-3.5 of Tate.

Contents:

- Preparation
- Exercises
  - Exercise 1: Message Information
  - Exercise 2: More Message Info
  - Exercise 3: A Colon Method
  - Exercise 4: Fun with Curly Brackets
  - Exercise 5: Actors
  - Exercise 6: Actors, Revisited
- For Those with Extra Time
  - Extra 1: XML

Preparation

a. Create a directory for the lab.

b. Verify that Io is installed on the workstation you’re using by typing `io` or `/usr/local/bin/io` at the command line. If Io is not installed on your workstation, you can ssh to church, where it is installed.

Exercises

Exercise 1: Message Information

The following method is intended to print out information on a message. Add it to your Io environment. (I would suggest that you save it in a file with a name like `callInfo.io` and use `launchFile` to load it.

```io
Object callInfo := method(
  "Message: " print;
  call message println;
  "Arguments: " print;
  call message arguments println;
  "Sender: " print;
  call sender println;
  "Target: " print;
  call target println;
)
```
a. How do you expect the Io interpreter to respond to the command Object callInfo?

b. Check your answer experimentally.

c. How do you expect the Io interpreter to respond to the command 3 callInfo? (Specify message, parameters, source, and target.)

d. Check your answer experimentally.

e. How do you expect the Io interpreter to respond to the command 3 callInfo(3, "hello", :goodbye)? (Specify message, parameters, source, and target.)

f. Check your answer experimentally.

g. How do you expect the Io interpreter to respond to the command callInfo (that is, with no object specified)?

h. Check your answer experimentally.

i. Summarize what you’ve learned from this exercise.

**Exercise 2: More Message Info**

Change the name of the slot from the previous command from callInfo to forward.

a. What do you expect the output of 3 forward to be?

b. Check your answer experimentally.

c. What do you expect the output of 3 foo to be?

d. Check your answer experimentally.

e. What do you expect the output of foo to be?

f. Check your answer experimentally.

g. As you may have just discovered, the change to the Object message forward method does not seem to affect the interpreter object. Fill in the forward slot of the interpreter object.

**Exercise 3: A Colon Method**

*This exercise is based on an example from Tate that is taken from an example originally by Jeremy Tregunna and updated by Chris Kappler.*

One of the characteristics of Io is that it’s easy to build new syntax. Let’s start with a simple : operation, which is intended to mean “store in a hash table”.
a. Add the handler for `:` with the following command.

```plaintext
OperatorTable addAssignOperator(":", "colon")
```

b. What do you expect to happen when you try to evaluate `3:4`?

c. Check your answer experimentally.

d. As you’ve noted, the colon method gets called, but we have not defined that method. So let’s define a preliminary version. Define a method, `colon`, that prints out its arguments. Then verify that it works okay with something like "Sam":"x4410".

e. When you printed out the arguments, you may have noticed that the evaluation of the colon method includes the quotation marks as part of the first string argument. We want to remove those marks. Write a method, `stripQuotes`, that takes a string as a parameter and returns the string with the opening quotation mark and closing quotation mark removed.

Note that you’ll need to make the string mutable with `asMutable` and that you can remove a prefix with `removePrefix` and a suffix with `removeSuffix`.

f. Of course, our real goal is to use the colon operator to fill in values in a `Map` object. Add a `colon` handler to `Map` that calls `atPut` appropriately.

g. Verify that the following works as you might expect:

```plaintext
info := Map clone
info "SamR" : "x4410"
```

Exercise 4: Fun with Curly Brackets

This exercise is based on an example from Tate that is taken from an example originally by Jeremy Tregunna and updated by Chris Kappler.

a. What do you expect to happen when you type `{ "Hello" }`?

b. Check your answer experimentally.

c. As you’ve noted, when you type an expression involving curly brackets, the interpeter object receives a `curlyBrackets` message. Define that message to print out its arguments.

d. Check what the output of the following is:

```plaintext
{
   "SamR" : "x4410",
   "Reg I. Strar" : "x3450"
}
```

d. As you may recall, for the phone book example, we’d like to “evaluate” the `"Name" : "Value"` pair for each such pair. Write a `curlyBrackets` handler for the `Map` object that calls `doMessage` with each argument.
Exercise 5: Actors

Try the actors example from Tate.

Exercise 6: Actors, Revisited

Consider the following method:

\[
\textit{countdown} := \text{method (list(5,4,3,2,1) foreach(x, x println))}
\]

a. What do you expect to happen when you type \textit{countdown}?

b. Check your answer experimentally.

c. What do you expect to happen when you type @@countdown?

d. Check your answer experimentally.

e. As you may have noted, you got no output. Why? Because the top level command finished before all of the actors. As the example suggests, you need to wait a bit to give the actors time to work. What do you expect to get when you type

\[
\texttt{Io}\texttt{> }@@\texttt{countdown}; \texttt{wait(0.5)}
\]

f. Check your answer experimentally.

You may have gotten a very strange answer the first time you tried this command. Try it again. And again. Have you figured out what’s going on yet?

h. Revise the method to read

\[
\textit{countdown} := \text{method (list(5,4,3,2,1) foreach(x, wait(0.2); x println))}
\]

What do you expect to get when you run the following?

\[
\texttt{Io}\texttt{> }@@\texttt{countdown}; \texttt{wait(0.5)}
\]

i. Check your answer experimentally. Then check it again. Explain what’s happening

For Those with Extra Time

Extra 1: XML

This exercise is based on an example from Tate.

As we noted earlier, by intercepting the \texttt{forward} message, we can alter Io’s behavior on undefined messages. We can change the effect of \texttt{forward} on all objects or simply on objects in a particular class (or lots of variants thereof).
a. Write an Io “class”, Builder0, with a forward handler that (i) prints out the name of the message and (ii) prints out the arguments to the message, one per line.

b. Write an Io “class”, Builder1, with a forward handler that (i) prints out the name of the message and (ii) recurses on the arguments. That is, it should send each argument to itself with \texttt{self doMessage(arg)}.

c. Write an Io “class”, Builder2, that works like Builder1 except that it checks the return value of \texttt{doMessage} and if it is non-nil, prints it out.

d. Write an Io “class”, Builder3, that (i) prints the message name as an opening tag, (ii) recurses on the arguments as in Builder2, and (iii) prints the message name as a closing tag.