CAP Assignment

Title: PROG

Course: CAP 2018-19 Q1

Students:

- Alexis Rico Carreto
- Pau Juliò Plana

Motivation

The objective behind the lab assignment for CAP is to build a basic instruction-oriented interpreter in Pharo 3.0. The interpreter must be able to store/read data into memory locations and perform incoditional jumps between scopes.

Assignment

Descripció resumida

tl;dr ⇒ Aquesta pràctica va de la pila de execució i la possibilitat de reificar-la.

La pràctica de CAP d'enguany serà una investigació del concepte general d'estructura de control, aprofitant les capacitats d'introspecció i intercessió que ens dóna Smalltalk.

Estudiarem les conseqüències de poder inspeccionar la pila d'execució (a la que tenim accés gràcies a la pseudo-variable thisContext) per fer-la servir i/o manipular-la.

Material a entregar

tl;dr ⇒ Amb l'entrega del codi que resol el problema que us poso a la pràctica NO n'hi ha prou. Cal entregar un informe i els tests que hagueu fet.

Haureu d'implementar el que us demano i entregar-me finalment un informe on m'explicareu, què heu aprés, i com ho heu arribat a aprendre (és a dir, m'interessa especialment el codi lligat a les proves que heu fet per saber si la vostra pràctica és correcte).

Les vostres respostes seran la demostració de que heu entés el que espero que entengueu. El format de l'informe és lliure, i el codi que m'heu d'entregar me'l podeu entregar de diverses maneres.

Table of Contents

- CAP Assignment
  - Motivation
  - Assignment
    - Descripció resumida
    - Material a entregar
  - Table of Contents
  - Development process
    - Unit and Integration initial tests
      - testCaseBase1
      - testCaseBase2
      - testCaseBinding1
      - testCaseBinding2
    - Utility functions
      - changeBinding: [Symbol]
      - storeLocalVariable:in: [Class side]
      - storeInstruction:in: [Class side]
      - storeBinding:with:in: [Class side]
    - Main program
      - withInit:do: [Class side]
      - initializeWith:and: [Instance side]
      - withInit:do: [Instance side]
    - Extra tests
      - testCaseEmptyInstructions
      - testCaseCapturedVariables
      - testCaseContinuation
      - testCaseDefaultReturn1
      - testCaseDefaultReturn2
      - testCaseEarlyReturn
      - testCaseInnerBlockReturn
      - testCaseJumping
      - testCaseUndefinedSymbol
  - Lessons learnt
  - Final conclusions

Development process

To develop the project we used a test-driven (TDD) approach. We will now explain:

1. The four tests that we used to build and validate the code.
2. The principles behind the auxiliar functions we built.
3. The structure of the main program.
4. The extra tests that we built to ensure the code quality.

Unit and Integration initial tests

testCaseBase1

Assignment base test case 1, performs a basic program execution.

```
testCaseBase1
    | result |

    result := PROG withInit: { #n . { #d . 0 } } do: {
        { #label1 . [ #n changeBinding: 0 ] } .
        { #label2 . [ #n changeBinding: (#n binding + 1) ] } .
        { #label3 . [ #n changeBinding: (#n binding + 1) ] } .
        { #label4 . [ #n changeBinding: (#n binding + 1) ] } .
        { #label5 . [ #n changeBinding: (#n binding + 1) ] } .
        { #label6 . [ #n changeBinding: (#n binding + 1) ] } .
        { #label7 . [ #RETURN binding value: #n binding  ] } .
    }.

    self assert: result == 5.
```

testCaseBase2

Assignment base test case 2, performs a program execution with some basic jumps.

```
testCaseBase2
    | result |
    result := PROG withInit: {{ #n . 10 } . { #coll . OrderedCollection new }}
do: {
        { #label1 . [ #n binding == 0 ifTrue: [ #RETURN binding value: #coll
binding ]  ] } .
        { #label2 . [ #coll changeBinding: ((#coll binding) add: #n binding;
yourself) ] } .
        { #label3 . [ #n changeBinding: (#n binding - 1) ] } .
        { #label4 . [ #label1 binding value ] }
    }.
    self assert: (result includesAll: #(10 9 8 7 6 5 4 3 2 1)).
```

testCaseBinding1

Small test case that unit tests changeBinding implementation of a symbol in a given block.

```
testCaseBinding1
    #n bindTo: 1 in: [
        #n changeBinding: 10.
        self assert: #n binding == 10.
    ] .
```

testCaseBinding2

Small test case that unit tests changeBinding implementation to detect the scenario of a not-known symbol.

```
testCaseBinding2
    self should: [
        #binding changeBinding: 10.
    ] raise: Error. "No binding for binding"
```

Utility functions

We built four different auxiliar functions to create the required binding functionality both to store new symbols and to update existing ones in a block scope.

changeBinding: [Symbol]

We basically re-used the implementaion of the *binding* method in *Symbol* class and changed the block that gets executed with the ifTrue clause. In the following snippet, we can observe the changes introduced in this new method.

```
Symbol methodsFor: '*PracticaCAP'
changeBinding: newValue
    | context |
    context := thisContext.
    [ context = nil ] whileFalse:
        [   ((context receiver isMemberOf: Binding)
                and: [ context selector = #of:to:in:
                    and: [context receiver key = self]])
                "Store new binding value and return"
                ifTrue: [^ context receiver value: newValue]
                ifFalse: [context := context sender]
        ].
    self error: 'No binding for ', self asString.
```

storeLocalVariable:in: [Class side]

We also added an utility method that given an OrderedCollection named initBlock, creates a binding of the first element and alters the OrderedCollection removing it.

Notice that local varaibles in initBlock can be represented either as a key-value pair or a key alone that will have nil value.

```
storeLocalVariable: initBlock in: aBlock
    | param |

    param := initBlock remove: (initBlock at: 1).

    param isArray ifTrue: [
        ^ self storeBinding: (param at: 1) with: (param at: 2) in: aBlock
    ].

    ^ self storeBinding: param with: nil in: aBlock.
```

storeInstruction:in: [Class side]

Similarly we added another utility method that given an OrderedCollection named instructionSet, creates a binding of the first element and alters the OrderedCollection.

Notice that all elements in the collection are key-value pairs.

Also notice that after evaluating the instruction source code we bind an incondition jump to the following instruction. If there's no following instruction we bind to the return directive (which by default will return nil as value).

```
storeInstruction: instructionSet in: aBlock
    | instruction nextSymbol |

    instruction := instructionSet remove: (instructionSet at: 1).

    nextSymbol := instructionSet isEmpty
        ifTrue: [ #RETURN ]
        ifFalse: [ (instructionSet at: 1) at: 1 ].

    ^ self storeBinding: (instruction at: 1) with: [
        (instruction at: 2) value.
        nextSymbol binding value
    ] in: aBlock
```

storeBinding:with:in: [Class side]

We added a wrapper method to interact with Symbol bindTo message using the adapter pattern.

```
storeBinding: aSymbol with: aValue in: aBlock
    aSymbol bindTo: aValue in: aBlock.
```

Main program

The main program consists of three different methods:

1. Static class-side entry point
2. Instance side initialization method
3. Recursive instance side main method

withInit:do: [Class side]

As the entry point for our program we defined a class-side method that receives the parameters and creates an instance that will control the whole execution.

Notice that this is the point we decided to create the Continuation that will be used as return point of the execution.

Also if the instructionSet parameter is empty we signal an error to the user, initBlock parameter can be empty though.

```
withInit: initBlock do: instructionSet
    instructionSet isEmpty ifTrue: [ Error signal: 'Empty Instruction Set' ].

    "Create the default return point as a continuation"
    ^ Continuation callcc: [ :kont |
        | instance |
        instance := self new.

        "Initialize instance with aContinuation to return and aSymbol to start"
        instance initializeWith: kont and: ((instructionSet at: 1) at: 1).

        "Launch recursive execution"
        instance withInit: (initBlock asOrderedCollection)
                 do: (instructionSet asOrderedCollection).
    ].
```

initializeWith:and: [Instance side]

In the instance side we decided to have the first instruction and the return continuation as instance variables. So we added an initialization method to allow the passing message.

```
initializeWith: aContinuation and: aSymbol
    finalDestination := aContinuation.
    entryPoint := aSymbol.
```

withInit:do: [Instance side]

Finally we created the main recursive method that does all the magic behind our program. It consists of two recursive cases and one base case.

If the initBlock is not empty we call #storeLocalVariable. It is guaranteed that this recursive case will finish because #storeLocalVariable removes the first element of initBlock with each call.

Similarly if the instructionSet is not empty we call #storeInstruction. It is guaranteed that this recursive case will finish because #storeInstruction removes the first element of instructionSet with each call.

Finally if both initBlock and instructionSet are empty we provide the base case which creates the #RETURN binding using the instance variable finalDestination (holding the return Continuation) and starts the execution with the first instruction using the label we stored on entryPoint instance variable.

```
withInit: initBlock do: instructionSet
    "If we have local variables left, store them as bindings"
    initBlock isEmpty ifFalse: [
        ^ self class storeLocalVariable: initBlock in: [
            self withInit: initBlock do: instructionSet.
        ].
    ].

    "If we have instructions left, store them as bindings"
    instructionSet isEmpty ifFalse: [
        ^ self class storeInstruction: instructionSet in: [
            self withInit: initBlock do: instructionSet
        ].
    ].

    "Store return point and start execution with first instruction"
    ^ self class storeBinding: #RETURN with: finalDestination
                in: [ entryPoint binding value ].
```

Extra tests

After building the main code we created a set of different integration tests with extreme scenarios. We detected a few issues and corrected them during this process. All the code found in this document is the final one and passes all the tests created.

testCaseEmptyInstructions

A test that verifies an error is signaled if no instructions are provided to PROG.

```
testCaseEmptyInstructions
    self should: [
        PROG withInit: { "Empty parameters" } do: { "Empty instructions" }.
    ] raise: Error.
```

testCaseCapturedVariables

A test that has a local variable outside the program execution but that we expect to be captured in the execution of PROG.

```
testCaseCapturedVariables
    | result capturedVariable |
    capturedVariable := 0.
    result := PROG withInit: { { #n . 0 } } do: {
        { #label1 . [ capturedVariable := 5 ] } .
        { #label2 . [ #RETURN binding value: capturedVariable ] } .
    }.

    self assert: result == 5.
```

testCaseContinuation

A test that has a local variable outside the program execution that will host a Continuation inside the program execution. We expect the Continuation to work properly inside PROG.

```
testCaseContinuation
    | result kont |
    result := PROG withInit: { "Empty variables" } do: {
        { #label1 . [ kont := Continuation callcc: [ :cc | cc ] ] } .
        { #label2 . [ (kont == 10) ifFalse: [ kont value: 10 ] ] } .
        { #label3 . [ #RETURN binding value: kont ] } .
    }.

    self assert: result == 10.
```

testCaseDefaultReturn1

A very simple test that verifies nil is returned when no other value has been provided to the return directive.

```
testCaseDefaultReturn1
    | result |

    result := PROG withInit: { "Empty variables" } do: {
        { #label . [ #RETURN binding value ] } .
    }.

    self assert: result isNil
```

testCaseDefaultReturn2

A very simple test that verifies nil is returned when no return directive has been provided.

```
testCaseDefaultReturn2
    | result |

    result := PROG withInit: { "Empty variables" } do: {
        { #label . [ "Empty block" ] } .
    }.

    self assert: result isNil.
```

testCaseEarlyReturn

A test that verifies any instruction can return and the program execution finishes properly with the received value.

```
testCaseEarlyReturn
    | result capturedVariable |

    capturedVariable := 0.

    result := PROG withInit: { "Empty variables" } do: {
        { #label1 . [ #RETURN binding value: 1 ] } .
        { #label2 . [ capturedVariable := 5 ] } .
        { #label3 . [ #RETURN binding value: 2 ] } .
    }.

    self assert: capturedVariable == 0.
    self assert: result == 1.
```

testCaseInnerBlockReturn

A test case where we verify inner blocks can also return to outside context and captured variables are shared across instructions.

```
testCaseInnerBlockReturn
    | result capturedVariable |
    capturedVariable := 0.

    result := PROG withInit: { "Empty variables" } do: {
        { #label1 . [ capturedVariable := [ capturedVariable := 5. #RETURN binding
value: 1 ] ] } .
        { #label2 . [ capturedVariable value ] } .
        { #label3 . [ #RETURN binding value: 2 ] } .
    }.

    self assert: capturedVariable == 5.
    self assert: result == 1.
```

testCaseJumping

A test case where we verify inconditional jumps between instructions are properly implemented and that when jumping, instructions found in the middle are completely ignored.

```
testCaseJumping
    | result |

    result := PROG withInit: { { #n . 0 } } do: {
        {#label1 . [ #label3 binding value ] } .
        {#label2 . [ #n changeBinding: 1 ] } .
        {#label3 . [ #RETURN binding value: #n binding ] } .
    }.

    self assert: result == 0.
```

testCaseUndefinedSymbol

A simple test to verify that an error is signaled if trying to bind to a symbol that does not exist.

```
testCaseUndefinedSymbol
    self should: [
        PROG withInit: { "Empty parameters" } do: {
            { #label1 . [ #label2 binding value ] } .
            { #label3 . [ #RETURN binding value ] } .
        }.
    ] raise: Error. "No binding for label2"
```

Lessons learnt

The main lesson we learnt is that building a classic "compiler" or "interpreter" is not only possible on high level languages but actually really easy to implement. Anyway, the main problem we observe is that not every single language out there support the level of inspection and instrospection to the call stack as SmallTalk.

The symbol binding as introduced in the BYTE magazine is really interesting and has allowed us to do all the implementation. Also as we were using standard methods within SmallTalk the rest of language specific features continued working as Continuations or inner block scope.

Final conclusions

The lab assignment has been really interesting and has expanded our views of how we can implement low level features on top of higher level languages. The project as a whole has been really satisfying and we did enjoy a lot building it.

Also this lab assignment has been the final task we needed not only to fully understand how SmallTalk works but to have real world experience in SmallTalk as a platform to build programs and applications.