Syntax and Messages

Stéphane Ducasse
Stephane.Ducasse@univ-savoie.fr
http://www.iam.unibe.ch/~ducasse/

Outline

- ***Literals: numbers, strings, arrays....***
- Variable, assignments, returns
- Pseudo-variables
- Message Expressions
- Block expressions
- Conditional and Loops

Originally Made for Kids

Read it as a non-computer-literate person:

```
| bunny |
bunny := Actor fromFile: 'bunny.vrml'.
bunny head doEachFrame:
   [ bunny head
         pointAt: (camera
                  transformScreenPointToScenePoint:
                           (Sensor mousePoint)
                                    using: bunny)
        duration: camera rightNow ]
```

Numbers

- SmallInteger, Integer,
  - 4, 2r100 (4 in base 2),3r11 (4 in base 3), 1232
- Automatic coercion
  - 1 + 2.3 -> 3.3
  - 1 class -> SmallInteger
  - 1 class maxVal class -> SmallInteger
  - (1 class maxVal + 1) class -> LargeInteger
- Fraction, Float, Double
  - 3/4, 2.4e7, 0.75d
  - (1/3) + (2/3) -> 1
  - 1000 factorial / 999 factorial -> 1000
  - 2/3 + 1 -> (5/3)

Characters

- Characters:
  - $F, $Q $U $E $N $T $i $N
- Unprintable characters:
  - Character space, Character tab, Character cr

Strings

- Strings:
  - #mac asString -> 'mac'
  - 12 printString -> '12'
  - 'This packet travelled around to the printer' 'I''idiot'
  - String with: $A
  - Collection of characters
  - 'lulu' at: 1 -> $l
- To introduce a single quote inside a string, just double it.

Symbols

- Symbols:
  - #class #mac #at:put: #+ #accept:
- Kinds of String
- Unique in the system (see after)

Symbols vs. Strings

- Symbols are used as method selectors, unique keys for dictionaries
- A symbol is a read-only object, strings are mutable objects
- A symbol is unique, strings are not

#calvin == #calvin -> true
'calvin' == 'calvin' -> false
#calvin, #zeBest -> 'calvinzeBest'

- Symbols are good candidates for identity based dictionaries (IdentityDictionary)

- Hint: Comparing strings is slower then comparing symbols by a factor of 5 to 10. However, converting a string to a symbol is more than 100 times more expensive.

Comments and Tips

- "This is a comment"
- A comment can span several lines. Moreover, avoid putting a space between the " and the first character. When there is no space, the system helps you to select a commented expression. You just go after the " character and double click on it: the entire commented expression is selected. After that you can printIt or doIt, etc.

Arrays

#(1 2 3) #('lulu' (1 2 3)) -> #('lulu' #(1 2 3))

- #(mac node1 pc node2 node3 lpr) an array of symbols.
- When one prints it it shows

#(#mac #node1 #pc #node2 #node3 #lpr)

- Byte Array   #[1 2 255]

Arrays

- Heterogenous

  #('lulu' (1 2 3)) -> #('lulu' #(1 2 3))
  #('lulu' 1.22 1) -> #('lulu' 1.22 1)
- An array of symbols:
  - #(calvin hobbes suzie) -> #(#calvin #hobbes #suzie)
- An array of strings:

  #('calvin' 'hobbes' 'suzie') -> #('calvin' 'hobbes' 'suzie')

Syntax Summary

comment:	“a comment”
character:	$c $h $a $r $a $c $t $e $r $s $# $@
string:	‘a nice string’ ‘lulu’ ‘l’’idiot’
symbol:	#mac #+
array:	#(1 2 3 (1 3) $a 4)
byte array:	#[1 2 3]
integer:	1, 2r101
real:	1.5, 6.03e-34,4, 2.4e7
float:	1/33
boolean:	true, false
point:	10@120

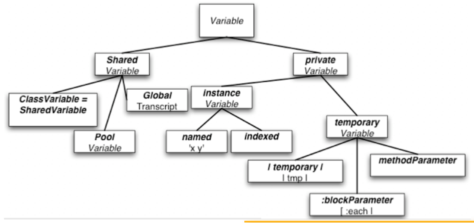
Note that @ is not an element of the syntax, but just a message sent to a number. This is the same for /, bitShift, ifTrue:, do: ...

Roadmap

- Literals: numbers, strings, arrays....
- ***Variable, assignments, returns***
- Pseudo-variables
- Message Expressions
- Block expressions
- Conditional and Loops

Variables

- Maintains a reference to an object
- Dynamically typed and can reference different types of objects
- Shared (starting with uppercase) or local (starting with lowercase)



Temporary Variables

- To hold temporary values during evaluation (method execution or sequence of instructions)
- Can be accessed by the expressions composing the method body.
  - **| mac1 pc node1 printer mac2 packet |**

Temporary Variable Good Style

- Avoid using the same name for a temporary variable and a method argument, an instance variable or another temporary variable or block temporary. Your code will be more portable. Do not write:

```
aClass>>printOn: aStream
    |aStream|
    ...
```

- Instead, write:

```
aClass>>printOn: aStream
    |anotherStream|
    ...
```

- Hint: Avoid using the same temporary variable for referencing two different objects

Assignments

- An Assignment is not done by message passing. It is one of the few syntactic elements of Smalltalk.

```
variable := aValue
three := 3 raisedTo: 1
variable1 := variable2 := aValue
```

- Avoid using var := var2 := var3
- To not try to know in which order the expressions is evaluated. You will write good code

Pointing to the Same Object

- In Smalltalk, objects are manipulated via implicit pointers: everything is a pointer.
- Take care when different variables point to the same object:

```
p1 := p2 := 0@100
p1 x: 100
p1 -> 100@100
p2 -> 100@100
```

Method Arguments

- Can be accessed by the expressions composing the method.
- Exist during the execution of the defining method.
- Method Name Example:
  accept: aPacket
- In C++ or Java:
  void Printer::accept(aPacket Packet)

Arguments are read-only

- Method arguments cannot change their value within the method body.
- Invalid Example, assuming contents is an instance variable:

```
MyClass>>contents: aString
    aString := aString, 'From Lpr'.
```

- Valid Example

```
MyClass>>contents: aString
    | addressee |
    addressee := aString , 'From Lpr'
```

Method Return

Use ^ expression to return the value of expression from a method

```
Rectangle>>area
    ^ width * height
```

By default ***self*** is returned

Instance Variables

- Private to a particular instance (not to all the instances of a class like in C++).
- Can be accessed by all the methods of the defining class and its subclasses.
- Has the same lifetime as the object.
- Declaration

```
Object subclass: #Node
        instanceVariableNames: 'name nextNode '
        ...
```

Instance Variables

- Scope: all the methods of the class

```
Node>>setName: aSymbol nextNode: aNode
            name := aSymbol.
            nextNode := aNode
```

- But preferably accessed using accessor methods

```
Node>>name
            ^name
```

Global Variables

- Always Capitalized (convention)
  - MyGlobalPi := 3.1415
- If it is unknown, Smalltalk will ask you if you want to create a new global
  - Smalltalk at: #MyGlobalPi put: 3.14
  - MyGlobalPi
    -> 3.14
  - Smalltalk at: #MyGlobalPi
    -> 3.14
- Stored in the default environment: Smalltalk in Squeak, VW has namespaces
- Design Hints: Accessible from everywhere, but it is not a good idea to use them

Roadmap

- Literals: numbers, strings, arrays....
- Variable, assignments, returns
- **Pseudo-variables**
- Message Expressions
- Block expressions
- Conditional and Loops

Six Pseudo-Variables

- Smalltalk expressions can contain true, false, nil, self, super thisContext, but cannot change their values. They are hardwired into the compiler.
- nil nothing, the value for the uninitialized variables. Unique instance of the class UndefinedObject

Six Pseudo-Variables

- **true**
  - unique instance of the class True
- **false**
  - unique instance of the class False
- Hint: Don't use False instead of false. false is the boolean value, False the class representing it. So, the first produces an error, the second not:
- False ifFalse: [Transcript show: 'False'] -> error
- false ifFalse: [Transcript show: 'False']

self, super, and thisContext

- Only make sense in a method body
- **self** refers to the receiver of a message.
- **super**
  - refers also to the receiver of the message but its semantics affects the lookup of the method. It starts the lookup in the superclass of the class of the method containing the super.
- **thisContext**
  - refers to the instance of MethodContext that represents the context of a method (receiver, sender, method, pc, stack). Specific to VisualWorks and to Squeak

self and super examples

```
PrinterServer>>accept: thePacket
   "If the packet is addressed to me, print it.
   Otherwise behave normally."

   (thePacket isAddressedTo: self)
      ifTrue: [self print: thePacket]
      ifFalse: [super accept: thePacket]
```

Roadmap

- Literals: numbers, strings, arrays....
- Variable names
- Pseudo-variables
- Assignments, returns
- **Message Expressions**
- Block expressions
- Conditional and Loops

Objects and Messages

- Objects communicate by sending message
- Objects react to messages by executing methods

Bot new go: 30 + 50

- A message is composed of:
- a receiver, always evaluated (Bot new)
- a selector, never evaluated #go:
- and a list possibly empty of arguments that are all evaluated (30 + 50)
- The receiver is linked with self in a method body.

Three Kinds of Messages

- Unary Messages

```
2.4 inspect
macNode name
```

- Binary Messages

```
1 + 2 -> 3
(1 + 2) * (2 + 3) -> 15
3 * 5 -> 15
```

- Keyword Messages

```
6 gcd: 24 -> 6
pcNode nextNode: node2
Turtle new go: 30 color: Color blue
```

Unary Messages

aReceiver aSelector

```
node3 nextNode  -> printerNode
node3 name      -> #node3
1 class         -> SmallInteger
false not       -> true
Date today      -> Date today September 19, 1997
Time now        -> 1:22:20 pm
Double pi       -> 3.1415926535898d
```

Binary Messages

aReceiver aSelector anArgument

- Used for arithmetic, comparison and logical operations
- One or two characters taken from:
  – + - / \ * ~ < > = @ % | & ! ? ,

```
1 + 2
2 >= 3
100@100
'the', 'best'
```

- Restriction:
- second character is never $-

Simplicity has a Price

- no mathematical precedence so take care

```
3 + 2 * 10 -> 50
3 + (2 * 10) -> 23
(1/3) + (2/3) and not
1/3 + 2/3
```

Keyword Messages

```
receiver
   keyword1: argument1
   keyword2: argument2

1 between: 0 and: 5
dict at: #blop put: 8+3
```

- In C-like languages it would be:

receiver.keyword1keyword2...(argument1 type1, argument2, type2) : return-type

Keyword Messages

```
Workstation **withName:** #Mac2
mac **nextNode:** node1
Packet
        **send:**  'This packet travelled around to'
        **to:**  #lw100
1@1 **setX:** 3
#(1 2 3) **at:** 2 **put:** 25
1 **to:** 10 -> (1 to: 10) anInterval
Browser **newOnClass:** Point
Interval **from:**1 **to:** 20  -> (1 to: 20)
12 **between:** 10 **and:** 20  -> true
x > 0 **ifTrue:**['positive'] **ifFalse:**['negative']
```

Composition Rules

- ***Unary-Msg > Binary-Msg > Keywords-Msg***
- at same level, from the left to the right

```
2 + 3 squared -> 11
2 raisedTo: 3 + 2 -> 32
#(1 2 3) at: 1+1 put: 10 + 2 * 3 ->  #(1 36 3)

2 raisedTo: 3 + 2 <=> (2 raisedTo: (3+2)) -> 32
```

Composition Rules

- **(Msg) >** Unary-Msg > Binary-Msg > Keywords-Msg

```
69 class inspect
(0@0 extent: 100@100) bottomRight
```

Hints ...

- Use () when two keyword-based messages
- occur within a single expression, otherwise the
- precedence order is fine.

```
x isNil ifTrue: [...]
```

- **isNil** is an unary message, so it is evaluated
- prior to **ifTrue:**

```
x includes: 3 ifTrue: [...]
```

- is read as the message includes:ifTrue:

```
(x includes: 3) ifTrue: [...]
```

- We use () to disambiguate them

Sequence

```
message1 .
message2 .
message3
```

. is a separator, not a terminator

```
| macNode pcNode node1 printerNode |
macNode := Workstation withName: #mac.
Transcript cr.
Transcript show: 1 printString.
Transcript cr.
Transcript show: 2 printString
```

For the Lazy: the Cascade

```
receiver
        selector1;
        selector2; ...
```

- To send multiple messages to the same object

```
Transcript show: 1 printString.
Transcript show: cr
```

- is equivalent to:

```
Transcript show: 1 printString ; cr
```

Syntax Summary

assigment: var := aValue	
unary message:	receiver selector
binary message:	receiver selector argument
keyword based:	receiver keyword1: arg1 keyword2:
arg2...	
cascade:	message ; selector ...
separator:	message . message
result:	^
parenthesis:	(...)

Roadmap

- Literals: numbers, strings, arrays....
- Variable, assignments, returns
- Pseudo-variables
- Message Expressions
- **Block expressions**
- Conditional and Loops

Blocks

- ***anonymous methods***
- ***deferred block of code***

fct(x) = x ^ 2 + x
fct (2) = 6
fct (20) = 420

```
|fct|
fct:= [:x | x * x + x].
fct value: 2         -> 6
fct value: 20        -> 420
fct                  -> aBlockClosure
```

Blocks Continued

```
[ :variable1 :variable2 |
    | blockTemporary1 blockTemporary2 |
    expression1.
    ...variable1 ... ]
```

- Two blocks without arguments and temporary variables

```
PrinterServer>>accept: thePacket
    (thePacket isAddressedTo: self)
    ifTrue: [self print: thePacket]
    ifFalse: [super accept: thePacket]
```

S.Ducasse

Block Evaluation

```
[....] value
    or value:                (for one arg)
    or value:value:          (for two args)
    or value:value:value:         …
    or valueWithArguments: anArray

[2 + 3 + 4 + 5] value
[:x | x + 3 + 4 + 5 ] value: 2
[:x :y | x + y + 4 + 5] value: 2 value: 3
[:x :y :z | x + y + z + 5] value: 2 value: 3 value: 4
[:x :y :z :w | x + y + z + w] value: 2 value: 3 value: 4 value: 5
```

S.Ducasse

Block

- The value of a block is the value of its last statement, except if there is an explicit return ^
- Blocks are first class objects.
- They are created, passed as argument, stored into variables...

S.Ducasse

Blocks - Continued

```
|index bloc |
index := 0.
bloc := [index := index +1].
index := 3.
bloc value -> 4

Integer>>factorial
  "Answer the factorial of the receiver. Fail if the receiver is
    less than 0."
    | tmp |
    ....
    tmp := 1.
    2 to: self do: [:i | tmp := tmp * i].
    ^tmp
```

S.Ducasse

- Literals: numbers, strings, arrays....
- Variable, assignments, returns
- Pseudo-variables
- Message Expressions
- Block expressions
- ***Conditional and Loops***

S.Ducasse

Yes ifTrue: is sent to a boolean

```
Weather isRaining
    ifTrue: [self takeMyUmbrella]
    ifFalse: [self takeMySunglasses]
```

ifTrue:ifFalse is sent to an object: a boolean!

S.Ducasse

Conditional: messages to booleans

- aBoolean **ifTrue:** aTrueBlock **ifFalse:** aFalseBlock
- aBoolean **ifFalse:** aFalseBlock **ifTrue:** aTrueBlock
- aBoolean **ifTrue:** aTrueBlock
- aBoolean **ifFalse:** aFalseBlock

```
(thePacket isAddressedTo: self)
    ifTrue: [self print: thePacket]
    ifFalse: [super accept: thePacket]
```

- Hint: Take care — true is the boolean value and True is the class of true, its unique instance!

S.Ducasse

Boolean Messages

- Logical Comparisons: &, |, xor:, not
  aBooleanExpr comparison ***aBooleanExpr***
  - (1 isZero) & false
  - Date today isRaining not
- Uniform, but optimized and inlined (macro expansion at compile time)
- aBooleanExpression ***or:*** orBlock
  orBlock will ***only*** be evaluated if aBooleanExpression is false
  false and: [1 error: 'crazy']
  -> false and not an error

S.Ducasse

Yes a collection is iterating on itself

```
#(1 2 -4 -86)
    do: [:each | Transcript show: each abs
printString ;cr ]

> 1
> 2
> 4
> 86
```

Yes we ask the collection object to perform the loop on itself

S.Ducasse

Some Basic Loops

- aBlockTest whileTrue
- aBlockTest whileFalse
- aBlockTest whileTrue: aBlockBody
- aBlockTest whileFalse: aBlockBody
- anInteger timesRepeat: aBlockBody

- [x<y] whileTrue: [x := x + 3]

- 10 timesRepeat: [Transcript show: 'hello'; cr]

For the Curious...

```
BlockClosure>>whileTrue: aBlock
  ^ self value
          ifTrue:[ aBlock value.
                  self whileTrue: aBlock ]

BlockClosure>>whileTrue
  ^ [ self value ] whileTrue:[]
```

For the Curious…

```
Integer>>timesRepeat: aBlock
    "Evaluate the argument, aBlock, the number of
    times represented by the receiver."

  | count |
  count := 1.
  [count <= self] whileTrue:
        [aBlock value.
       count := count + 1]
```

Choose your Camp!

To get all the absolute values of numbers you could write:

```
|result|
aCol :=  ( 2 -3 4 -35 4 -11).
result := aCol species new: aCol size.
1 to: aCollection size do:
     [ :each | result
                 at: each put: (aCol at: each) abs].
result
```

Choose your Camp (II)

- You could also write:

```
( 2 -3 4 -35 4 -11) collect: [ :each | each abs ]
```

- Really important: Contrary to the first solution, the second solution works well for indexable collections and also for sets.

Iteration Abstraction: do:/collect:

aCollection do: aOneParameterBlock
aCollection collect: aOneParameterBlock
aCollection with: anotherCollection do: aBinaryBlock

```
(15 10 19 68) do:
     [:i | Transcript show: i printString ; cr ]

(15 10 19 68) collect: [:i | i odd ]
   ->  (true false true false)

(1 2 3) with:  (10 20 30)
       do: [:x :y| Transcript show: (y ** x) printString ; cr ]
```

Opening the Box

Iterators are messages sent to collection objects
Collection is responsible of its traversal

```
SequenceableCollection>>do: aBlock
   "Evaluate aBlock with each of the receiver's elements
   as the argument."

   1 to: self size do: [:i | aBlock value: (self at: i)]
```

select:/reject:/detect:

aCollection select: aPredicateBlock
aCollection reject: aPredicateBlock
aCollection detect: aOneParameterPredicateBlock
aCollection
detect: aOneParameterPredicateBlock
ifNone: aNoneBlock

```
(15 10 19 68) select: [:i|i odd] ->  (15 19)
(15 10 19 68) reject: [:i|i odd] ->   (10 68)
(12 10 19 68 21) detect: [:i|i odd]  -> 19
(12 10 12 68) detect: [:i|i odd] ifNone:[1] -> 1
```

inject:into:

aCollection inject: aStartValue into: aBinaryBlock

```
| acc |
acc := 0.
(1 2 3 4 5) do: [:element | acc := acc + element].
acc
-> 15
```

Is equivalent to

```
  (1 2 3 4 5)
     inject: 0
     into: [:acc :element| acc + element]
  -> 15
```

Do not use it if the resulting code is not crystal clear!

Other Collection Methods

aCollection includes: anElement
aCollection size
aCollection isEmpty
aCollection contains: aBooleanBlock

```
(1 2 3 4 5) includes: 4 -> true
(1 2 3 4 5) size -> 5
(1 2 3 4 5) isEmpty -> false
(1 2 3 4 5) contains: [:each | each isOdd] -> true
```

What we saw

- Numbers (integer, real, float…), Character $a, String 'abc', Symbols (unique Strings) #jkk,
- Arrays (potentially not homogenous) #(a #(1 2 3), Array with: 2+3
- Variables:
  - Lowercase => private
  - Instance variables (visible in by all methods), method arguments (read-only), local variable |a|
  - Uppercase => global
- Pseudo Var: true, false, nil, self, super
  - self = **always** represents the msg receiver
  - nil = undefined value

What we saw

- Three kinds of messages
  - Unary: Node new
  - Binary: 1 + 2, 3@4
  - Keywords: aTomagoshi eat: #cooky furiously: true
- (Msg) > unary > binary > keywords
- Same Level from left to right
- Block
  - Functions
    fct(x)= x*x+3, fct(2).
    fct :=[:x| x * x + 3]. fct value: 2
  - Anonymous method
  - Passed as method argument:

```
factorial
 tmp:= 1.
 2 to: self do: [:i| tmp := tmp * i]
```