# Macquarie University School of Computing

# COMP3000 Programming Languages 2022

## Assignment 1 - Interpreter

Due: see submission page
Worth: 10% of unit assessment

Marks breakdown:

Code: 90%
Test: 10%

Submit a notice of disruption via ask.mq.edu.au if you are unable to submit on time for medical or other legitimate reasons.

Late penalty without proper justification: 10% of the full marks for the assessment per day or part thereof late; mark of 0% given after five days.

Unless a Special Consideration request has been submitted and approved, a 5% penalty (of the total possible mark) will be applied each day a written assessment is not submitted, up until the 7th day (including weekends). After the 7th day, a grade of '0' will be awarded even if the assessment is submitted. A 1-hour grace period is provided to students who experience a technical concern.

## Overview

The assignment code bundle provides a Scala skeleton program for constructing an interpreter for a simplified APL. For example, the inputs:

| | |
|---|---|
| `3 + 5` | produces 8 |
| `2 × 3` | produces 6; there is a special multiply symbol |
| `2 * 3` | produces 8; "*" means exponentiation |
| `3 × 4 + 5` | produces 27 ( = 3 × (4 + 5)); no precedence; just right to left |
| `3 - 2` | produces 1; right to left but functions aren't done backwards |
| `⍳ 4` | produces vector 1 2 3 4 |
| `+/⍳ 4` | produces 10 ( = 1+2+3+4) |
| `2 + 7 3 4` | produces the vector 9 5 6; adds 2 to each number in 7 3 4 |

If you want to see more about APL then visit

- https://en.wikipedia.org/wiki/APL_(programming_language)
- https://en.wikipedia.org/wiki/APL_syntax_and_symbols

## Language basics

As can be seen above, the language has a very simple syntax. While APL handles numbers and characters, we will do just numbers. There is no distinction between an integer and a floating-point number and the numbers are stored as doubles but displayed without decimal point when extremely close to an integer value.

As there is no Boolean data type, the numbers 1 and 0 are used to represent true and false respectively.

### *Data items*

The data items that can be manipulated are:

- scalar – a simple number, e.g. 3; negative numbers are written with a raised minus sign, e.g. ¯3
- vector – an array of numbers, e.g. 7 2 4; no notation is required to form a literal vector, just numbers next to each other
- matrix – a two-dimensional (rectangular) array of numbers; typically a matrix is created by performing a function on a vector (see later)

### *Monadic and dyadic functions*

APL has two types of functions:

- monadic (prefix), of the form: fn arg; e.g. ⌊ 4.7 gives 4 (floor of 4.7 is 4)
- dyadic (infix), of the form: a fn b; e.g. 3 + 4

Some function symbols can be used monadically and dyadically. If the left argument is present then the dyadic interpretation will always be chosen. For example, ⌊ means:

- floor when used monadically, e.g. ⌊ 4.7 gives 4
- but minimum when used dyadically, e.g. 2 ⌊ 4 gives 2

So, in:

```
2 + ⌊ 4.7          (gives 6)
```

there is no left argument for ⌊ ,so it is applied monadically, whereas in:

```
2 + 3 ⌊ 4.7        (gives 5)
```

there is a left argument (just 3, not 2 + 3).

### *Right to left*

APL has no operator precedence – execution starts at the right end of a line, e.g. in

```
3 × 4 + 5
```

the addition is done before the multiplication. To get the multiplication to be done first, then:

```
(3 × 4) + 5
```

Note the difference between the two minus signs:

```
¯2 + 3   is   1          -2 + 3   is   ¯5
```

In the first addition, it is (negative 2) plus 3; in the second, `- (2 + 3)`.

### *Function descriptions*

Here are all the functions we will be dealing with; a and b represent the left and right arguments:

| Symbol | Monadic use | Dyadic use | Our |
|---|---|---|---|

| | | | **name** |
|---|---|---|---|
| + | + b is the same as 0 + b | a + b is normal addition | |
| − | − b is the same as 0 - b | a – b subtraction | |
| × | × b is the sign of b (-1, 0 or 1)<br>× 4 is 1 | a × b multiplication | mul |
| ÷ | ÷ b is the reciprocal of b | a ÷ b division | div |
| * | * b is e to the b power | a * b a to the b power [NOT NEEDED] | |
| ! | ! b is factorial b<br>! 4 is 24 | a ! b binomial coefficient C(b, a)<br>2 ! 5 is 10 [NOT NEEDED] | |
| \| | \| b is the absolute value of b | a \| b b % a | |
| < | | a < b condition test [NOT NEEDED]<br>4 < 8 is 1 4 < 2 is 0 | |
| ≤ | | a ≤ b condition test [NOT NEEDED] | <= |
| = | | a = b condition test | |
| ≥ | | a ≥ b condition test [NOT NEEDED] | >= |
| > | | a > b condition test [NOT NEEDED] | |
| ≠ | | a ≠ b condition test | != |
| ~ | ~ b is logical NOT of b<br>~ 1 is 0 ~ 0 is 1 | | |
| ∨ | | a ∨ b logical OR [NOT NEEDED]<br>1 ∨ 0 is 1 | or |
| ∧ | | a ∧ b logical AND [NOT NEEDED]<br>0 ∧ 1 is 0 | and |
| ⌊ | ⌊ b is floor of b<br>⌊ 5.9 is 5 | a ⌊ b minimum<br>7 ⌊ 3 is 3 | flr |
| ⌈ | ⌈ b is ceiling of b | a ⌈ b maximum | ceil |
| ⍳ | ⍳ b is vector of numbers from 1 to b<br>⍳ 5 is the vector 1 2 3 4 5 | a ⍳ b index of b in a<br>8 5 3 ⍳ 5 is 2 (indexing starts from 1)<br>8 5 3 ⍳ 7 is 4 (1+⍴a means not found) | iota |
| ∈ | | a ∈ b is a in b?<br>4 ∈ 8 5 3 is 0 (false) | mem |
| ⍴ | ⍴ b is the shape of b (see later) | a ⍴ b reshapes b according to a (see later) | rho |
| ↑ | | a ↑ b takes the first a elements of b<br>2 ↑ 4 5 6 7 9 is 4 5<br>¯2 ↑ 4 5 6 7 9 is 7 9 | take |
| ↓ | | a ↓ b drops the first a elements of b<br>2 ↓ 4 5 6 7 9 is 6 7 9<br>¯2 ↓ 4 5 6 7 9 is 4 5 6 | drop |
| / | | a / b a selects elements of b<br>1 0 1 1 / 4 2 7 9 is 4 7 9 | |
| , | , b is b turned into a vector<br>, 7 is the vector 7 (not the scalar 7)<br>, applied to vector is just the vector<br>, applied to matrix is vector of rows of matrix concatenated | a , b concatenates two vectors [NOT NEEDED] | |

Functions marked "[NOT NEEDED]" are outside the scope of this assignment, including monadic and dyadic forms. They are there just for interest.

Each of the scalar functions can be applied element-by-element to vectors and matrices; e.g.

```
3 + 8 2 4        gives   11 5 7
9 1 1 + 4 8 8    gives   13 9 9
⌊ 3.7 9.3 7.8    gives   3 9 7
4 8 3 > 5 2 1    gives   0 1 1
```

The allowable combinations, using:

- S means scalar
- V means vector
- M means matrix
- A means any of the above

are:

| Function f | Monadic | Shape of monadic result | Dyadic | Shape of dyadic result |
|---|---|---|---|---|
| + - × ÷ * ! ⌊ ⌈ | f A | same as argument | S f S, S f V, V f S, S f M, M f S<br>V f V providing vectors same length<br>M f M providing matrices have same shape | shape of the argument that has the higher dimension |
| < ≤ = ≥ > ≠ ∨ ∧ | | | as above | as above |
| ~ | ~ A | same as argument | | |
| ⍳ | ⍳ S | vector of length S | V ⍳ A | shape of right argument |
| ∈ | | | A ∈ A | shape of left argument |
| ⍴ | ⍴ A | vector of length that is number of dimensions of A (scalar has 0 dimensions) | S ⍴ A, V ⍴ A | shape determined by left argument |
| ↑↓ | | | S f V | vector that is some part of right argument |
| , | , A | S vector 1<br>V same as for V<br>M vector of num elements in M | S,S S,V V,S V,V | vector of total length |

Also, any place that a scalar is accepted, a vector of length one is also accepted and the number in that vector is treated like the scalar. For example, monadic iota expects a scalar argument. The result of take is always a vector.

```
      ⍳ 1↑3 7 2
```

The result of the take is the vector 3 (which is indistinguishable from the scalar 3 when printed out) but iota is still able to use this as its argument, treating it like a scalar and giving the result 1 2 3.

### *Non-symbolic function names*

As you can see by now, APL has many symbols that you can't find on your keyboard. So, instead we will use the names given in the last column above. For example, to multiply two numbers:

```
2 mul 3
```

For simplicity, we will restrict all names in the language to just alphabetic rather than the usual alphanumeric. This means we can write the above, if we want, as:

```
2mul3
```

To write a negative number, we can't type `¯2`; instead we will use an underscore: `_2`.

### *Shape*

All data items have shape that is described by a vector:

| Data type | Shape |
|---|---|
| scalar | vector of length zero |
| vector | vector of length one containing the length of the original vector |
| matrix | vector of length two that contains the number of rows and columns of the matrix |

Dyadic rho can be used to create a vector or matrix of a particular size (left argument) and it is filled by repeatedly cycling through the right argument. For example:

```
4 ρ 7           gives     7 7 7 7
4 ρ 7 8 3       gives     7 8 3 7
2 2 ρ 1 2 3 4 5 gives     1 2
                          3 4
2 3 ρ 4 7 1 9   gives     4 7 1
                          9 4 7
```

rho applied to each of those four results would produce a vector containing the left argument of each of the four rhos above; e.g.

```
ρ 2 3 ρ 4 7 1 9  gives    2 3
```

### *Operators*

Operators are used to apply a function over a vector or matrix.

#### *Reduction*

The reduce operator (/) has the form:

```
f / b
```

(f is a function, b is a data item)

For example:

```
+ / 3 2 6 4
```

which calculates:

```
((3 + 2) + 6) + 4      giving  15
```

To find the maximum value in a vector:

```
⌈ / 3 2 6 4            giving  6
```

The operator is reducing a vector to a number. When applied to a matrix, it will reduce each row to a number and the overall result will be a vector; e.g.

```
+ / 2 2⍴1 2 3 4        giving  3 7
```

The operator ⌿ (name: “rdc”) is the same as / when applied to vectors but when working on a matrix, it reduces the columns rather than the rows. For example:

```
+ ⌿ 2 2⍴1 2 3 4        giving  4 6
```

### *Inner product*

Inner product is inspired by matrix multiplication. It has the form:

```
a f . g b
```

(f, g are functions; a, b are vectors or matrices)

For example:

```
a +.× b
```

This does matrix multiplication. Each row of a is combined, element by element, with each column of b using function g; and the vector result of combining a row of a with a column of b is reduced using function f with the resultant value becoming the value in the matrix result at that row,column position. (To make sense of this, just think about what happens in matrix multiplication using multiplication and addition and substitute functions g and f instead.)

### *Outer product*

Outer product has the form:

```
a ∘.f b
```

(f is a function; a, b are usually vectors)

(Name for ∘ is “out”.)

The result is a matrix in the same way that a multiplication table is formed. E.g.

```
2 3 4 ∘.+ 2 5    gives      4 7
                            5 8
                            6 9
```

## *Variables*

The language has variables. These do not need to be (and cannot be) declared. A value is given to a variable using a left-pointing arrow, e.g.

```
a ← 3
```

(written as “<-“)

An assignment expression (like above) has a value, which is the value that was assigned. So:

4 + a ← 3

has the value 7.

For our purposes, all variables are global.

User-defined functions are beyond the scope of this assignment.

## Technical description of underlying data structures

The tokens in our language are:

- "(" - a left bracket
- ")" - a right bracket
- a number (integer), e.g 123 or _78 (meaning negative 78); it is just made of digits with an optional "_" at the front for a negative number
- a symbol

A symbol can be:

- a string consisting of letters (e.g. "fred")
- a string of one or two non-alphanumeric characters (e.g. ">=")

These will be recognised using regular expressions as seen in an early class. See the function matchPat; you just have to supply the regular expression.

Because of the simplified approach to lexical analysis, other characters will essentially be ignored, e.g. @#$%^&[]{}:;"',. You can ignore all these and just assume that input will be reasonable.

Note that all uppercase letters are forced to lowercase by function matchPat.

Classes have been provided for the data objects in the language:

```
sealed abstract class AObject
case class ASymbol(sym:String) extends AObject
case class AOperator(op:String) extends AObject
case class ANumber(num:Double) extends AObject
case class AVector(v:Array[Double]) extends AObject
case class AMatrix(m:Array[Array[Double]]) extends AObject
case object LRBrac extends AObject
case object RRBrac extends AObject
case object Assign extends AObject
```

The tokens on the input line can be in any of the above classes except for AMatrix:

- Assign is for "<-"
- LRBrac is for "("
- RRBrac is for ")"
- ANumber is for a number by itself
- AVector is for a vector of numbers (i.e. numbers next to each other)
- AOperator is for all the listed function and operator symbols listed above (using our name if the APL symbol is not on the keyboard); note that this includes alphabetic names (e.g. "take") and keyboard symbols (e.g. "+")

- ASymbol is for everything else (which will be alphabetic names that are not known – these will be essentially variable names)

Once you have constructed a regular expression to recognise all the tokens, you will need to write the functions

- strToAObj which converts a token string to an AObject (i.e. one of the above bullet points); this function can assume that the regular expression handling worked so that, for example, if a token string starts with a digit then it must be all digits and can be converted to a double
- tokensToAObjs which takes a list of strings (tokens) and turns it into an (Option) list of AObject.

Error handling for input strings will be very basic. If a function is meant to return an AObject but there is an error then return err("some error message"). The source file has the err function for this purpose. For example if function strToAObj encounters a string token it doesn't recognise then it can return this error.

Once tokensToAObjs has produced something sensible, then it can be passed to function exec (as seen in Main.scala). Note that exec reverses the list of token objects so that eval can more easily process the list – the right end of the list is now the head of the list and the list is much easier to traverse.

A simple function that assists with evaluation is eval1 which takes a single object and determines its value.

Another useful function that assists with evaluating expressions is evalWithRightArg. For example, in lines that looks like these:

```
… 4
… 3 + 4
```

In the first case, the 4 is encountered, evaluated (to itself) and passed as the rightArg to evalWithRightArg. In the second case, after 3+4 is evaluated to 7, 7 is passed as the rightArg to evalWithRightArg. evalWithRightArg's job is to look at the next token or two and determine what to do next.

## Numbers

As all numbers are stored as doubles, there are various times doubles need to be compared or treated as integers. APL allows a fuzz factor so that if two doubles are very close then they are considered equal. There are several functions to assist with this:

- isNumObj – determines whether an object is a number object
- isInt – determines whether a double can be considered an integer (e.g. 3.2 no, 3.0 yes)
- getInt – converts a double to an int
- areEqual – determines whether two doubles are sufficiently close together to be considered equal

For example, "iota 37 div 10" should give an error because 3.7 is not an integer.

## Error handling

If an error is encountered during evaluation, then call the err function; e.g.

```
err("iota requires integer")
```

The function returns ASymbol("ERROR <your error message>") which is a valid return value during evaluation.

## Testing

Some test cases are supplied. You will need to write more test cases to check your functions.

Your test cases should be added at the end of src/test/scala/InterpretTests.scala after the TO DO comment.

Note: testing will not include extreme cases.

## What you have to do

The skeleton code bundle contains:

- Main.scala (in src/main/scala); no changes required to this (although you may want to do some informal testing from the main function
- Interpret.scala (in src/main/scala); flesh out the skeleton functions here
- InterpretTests.scala (in src/test/scala); add your test cases here (in this file at the bottom, not a separate file)

In Interpret.scala you will need to implement the required functionality. You will need to do the following (and these are mirrored with simple tests in the tests file).

### *Tokens [pass level]*

- token pattern matching – supply appropriate regular expressions for pat
- get strToAObj working – it converts a string token to the appropriate AObject
- get tokensToAObjects working – it converts a list of token strings to a list of Aobject's

### *Simple evaluation [pass level]*

- the four basic arithmetic functions (+, -, mul, div) with scalar arguments; monadic and dyadic
- other scalar functions with scalar arguments; monadic and dyadic
- iota; monadic and dyadic
- member
- rho; monadic and dyadic
- take, drop

### *Mixed shape [credit level]*

- allowing scalar and vector for function arguments (monadic and dyadic)
- allowing vector and vector for dyadic function arguments

### *Variables [credit level]*

- using variables

### *Operators [distinction/high distinction level]*

- reduce on vectors
- reduce ("/" and "rdc") on matrices
- outer product
- inner product

### *Full evaluation [distinction/high distinction level]*

- complex expressions with brackets

- complex expressions

## Functional programming

While the aim of the programming is to write the interpreter, you will also be assessed on how well your program is written in the style of functional programming. For example:

- use val instead of var for variable declarations
- try to avoid iterative loops (for/while); use map, filter, reduce, …
- use match instead of multiple Ifs
- the machinery for handling monadic functions is the same for most of the functions – it should be abstracted rather than repeated for each monadic function
- likewise for handling dyadic functions

## Running the program

The skeleton for this assignment is designed to be run from within sbt. In the command prompt for sbt, there are four relevant commands:

- test – runs the test suite (tests are in src/test/scala/iInterpretTests.scala
- run – executes the main function in src/main/scala/Main.scala which repeatedly prompts you to enter an APL expression to be executed; an empty input line will terminate it
- compile – compiles any changed source files; there is no need to separately compile because the run and test commands always do any required compilation
- console – this is used for impromptu testing (after you have done a compile); it will bring up a "scala>" prompt; then enter:

  import org.mq.interpret.Interpret._

  now you will be able to access the Scala functions and classes in Interpret.scala; e.g.

  exec(List(ANumber(3.0), AOperator("+"), ANumber(4.0))

## What you must hand in and how

A zip file containing:

- Interpret.scala
- InterpretTests.scala

Do not include any directories.

Your submission should include all of the tests that you have used to make sure that your program is working correctly. Note that just testing one or two simple cases is not enough for many marks. You should test as comprehensively as you can.

**Submit your code electronically as a single zip file called ass1.zip** using the appropriate submission link on the COMP3000 iLearn web page by the due date and time. (It is not necessary to include your name anywhere, but if you do want to include your name then put it as a comment at the top of the Scala files, NOT in the name of the zip file.)

DO NOT SUBMIT YOUR ASSIGNMENT IN ANY OTHER FORMAT THAN ZIP containing just those two .scala files. Use of any other format slows down the marking and may result in a mark deduction.

## Marking

The assignment will be assessed according to the assessment standards for the unit learning outcomes.

Your code will be assessed for correctness and quality with respect to the assignment description. Marking will also assess the adequacy of your test cases – this will be 10% of the marks for the assignment.

7 Aug 2022