APL – A Programming Language

António Menezes Leitão

1 Introduction

APL (**A P**rogramming **L**anguage) is one of the oldest programming languages, having been invented by Kenneth Iverson in 1957. The first implementation of the language was completed in 1964.

Just like the Lisp language (a predecessor of Common Lisp), APL was idealized as a mathematical language without any concern regarding the architecture of the computers where it would be executed. Although the original APL is, nowadays, not widely used, there are a miriad of APL dialects and descendants that are still used, including APL2, A+, APL2000, Dyalog APL, APL Sharp, MicroAPL, J, K, Nial, Glee, and NESL. There are also scientific computing languages such as Matlab or Octave that were strongly inspired by APL.

Just like Lisp, APL language is an interactive language, in the sense that it provides a way to interact with the programmer based on the evaluation of expressions provided by the programmer.

APL is characterized by being a language for processing arrays. An array is simply a group of values of the same type arranged in a rectangular structure. In mathematical terms an array is called a *tensor* and it has, as particular cases, scalars, vectors, and matrices. Through the use of arrays, it is possible to operate over multiple values simultaneously, dispensing the traditional control structures (selection, iteration and recursion) that are necessary in languages that can only handle a value at a time.

APL offers a large range of operations over arrays. In addition to the traditional mathematical operations on scalars, vectors, and matrices (sum, product, inner product, outer product, transposition, etc) there are numerous operations to create and adapt arrays (change the shape, change the number of dimensions, search, index, and remove elements, etc). In the original language, most of these operations are described using mathematical symbols (ρ, ι, $\times$, $\Box$, etc.).

By simultaneously manipulating large groups of values, APL operations provide considerable expressive power. This, together with the use of mathematical symbols, allows APL programs to be extremely succinct. For example, here is a **complete** APL program that finds all the prime numbers below a certain limit R:[1]

$$\sim R \in R \circ.\times R)/R \leftarrow 1 \downarrow \iota R$$

Despite being a very powerful language, APL has a tiny set of users. This happens, first, because learning the language is an intellectual challenge that not all developers are willing to take and, secondly, because the original language

[1]You don't need to understand the program at this moment.

uses a set of mathematical symbols that are not easy to write on traditional keyboards. To overcome the latter problem, some versions of the APL language allow mathematical symbols written in full (i.e., using `rho` instead of ρ, `iota` instead of ι, etc.) or use names that better describe the meaning of the operation (for example, using `drop` instead of $\downarrow$ and `interval` instead of ι).

In the following sections we provide an introduction to the language and we describe the primitive operations with greater detail.

2 APL Tutorial

For a more interactive introduction to the APL language, we suggest that you check the URL `http://tryapl.org/`.

2.1 Syntax and Semantics

APL handles scalars, i.e., single values, or arrays, i.e., grouped values. All these values are handled using **functions** (which includes the traditional mathematical operations addition, subtraction, etc.) and **operators**, that are functions that manipulate other functions.

Expressions are evaluated **from right to left** and, although parenthesis allow changing the order of evaluation, **all functions have the same precedence** (but **operators have more precedence than functions**). This makes the expression `5 * 6 - 2` evaluates to `20` (and not `28`), as it is treated as `5 * (6 - 2)`. For the same reason, `5 - 7 - 2` evaluates to `0` (and not `-4`). The order of evaluation can be rephrased by saying that the argument to the right of a function includes everything that is on his right.[2]

In APL, the functions (including traditional mathematical operators) can be of three types:

Niladic are functions that do not take arguments. They will not be discussed in this text.

Monadic are functions that take only one argument. The invocation syntax is of the form `function argument`. For example, the factorial of the number `5` is written `! 5`. In the case of operators, the invocation syntax is `argument operator`. For example, in the expression `+ fold`, the operator is `fold` and the argument is `+`.

Dyadic are functions that take two arguments. The syntax invocation is of the form `argument1 function argument2`. For example, the sum of the numbers `2`, and `3` is written `2 + 3`. Dyadic operators use the same notation.

In terms of elementary data types, APL is characterized by having only boolean, character, integer and floating point numbers. However, these types may be structured in *tensors*, and these may have any number of dimensions, althought there are some particular cases:

- When the tensor has zero dimensions, we have a *scalar*.

[2]This evaluation rule was chosen because APL has hundreds of operators and, contrary to traditional mathematics, it was impossible for a programmer to remember all the precedences involved in such a large number of operators.

- When the tensor has one dimension, we have a *vector*.
- When the tensor has two dimensions, we have a *matrix*.

Note that, in any case, all elements of a tensor are of the same type. This means that you can not mix in the same tensor elements of different types, for example. to create a vector that contains both characters and integers.

The available functions can be characterized in following categories:

Scalar functions are functions that take tensors as arguments and operate on all the elements of the tensor, but without changing the size (i.e., dimensions) or shape (i.e., the number of dimensions) of the tensors.

Restructuring functions are functions which change the size or the shape of tensors.

Mixed functions are functions that combine the two previous effects.

Operators are higher-order functions that produce functions from other functions.

Note that a function that accepts a tensor as argument also accepts a scalar, as a scalar is a particular case of a tensor that has zero dimensions.

2.2 Arithmetic Operations

APL provides the same arithmetic operations provided by other programming languages. As an example, consider the following interaction with the APL interpreter:

```
APL> 1 + 3
4
```

The text `APL>` is the *prompt* of the interpreter, indicating that it is expecting to receive an expression to evaluate.

Here are some more examples:

```
APL> 6 / 3
2
APL> 5 / 2
5/2
APL> 5.0 / 2
2.5
APL> 5 - 2
3
APL> 5 * 2
10
```

One of the most interesting features of APL is the fact that it deals as easily with scalars as it does with vectors and matrices. For example, we can add the vectors $[1, 2, 3] + [4, 5, 6]$ using:

```
APL> 1 2 3 + 4 5 6
5 7 9
```

It is also possible to perform operations between scalars, and vectors or matrices:

```
APL> 1 + 1 2 3 4 5
2 3 4 5 6
APL> 10 9 8 7 6 5 - 5
5 4 3 2 1 0
```

1APL

2.3 Assignment

In order to save intermediate results, APL allows assignments, via the assignment operator `=:`. Here are a few examples:

```
APL> pi =: 3.14159
3.14159
APL> raio =: 5
5
APL> p =: 2 * pi * raio
31.4159
APL> p =: 20 30 40 50
20 30 40 50
APL> p
20 30 40 50
APL> 1 + (n =: 99)
100
APL> n
99
```

2.4 Relational Operations

In addition to arithmetic, APL also allows the use of relational operations. However, we must take into account that the logical true and false values are, respectively, `1` and `0`, and that, for the purposes of logical operations, any value other than zero is taken as true. As a result, we have:

```
APL> 2 > 3
0
APL> 3 < 5
1
APL> x =: 1 2 3 4 5 6 7
1 2 3 4 5 6 7
APL> x > 3
0 0 0 1 1 1 1
APL> 5 > x
1 1 1 1 0 0 0
```

2.5 Higher-Order Operations

APL also provides *operators*, i.e., functions that accept functions as arguments. One of the most important operator is `fold`, also called *insertion* or *reduction*. This operator injects the function on his left between every pair of elements of the argument on its right. Examples:

```
APL> + fold 1 2 3 4
10
APL> 1 + 2 + 3 + 4
10
APL> * fold 1 2 3 4
24
APL> 1 * 2 * 3 * 4
24
```

In fact, operators are just special functions that modify the behavior of other functions. Operators receive functions as arguments and produce functions as results. Note that operators have higher precedence than functions and, thus, the expression `+ fold 1 2 3 4` is recognized by APL as `(+ fold) 1 2 3 4`.

The operator `scan` works similarly to the operator `fold`. Although `+ fold` calculates the sum of a vector of numbers, `+ scan` calculates the partial sums of the vector, i.e.:

```
APL> + scan 1 2 3 4
1 3 6 10
```

The `scan` operator can be seen as the application of the operator `fold` to successively larger fragments of the vector, as can be seen in the following example:

```
APL> + fold 1
1
APL> + fold 1 2
3
APL> + fold 1 2 3
6
APL> + fold 1 2 3 4
10
```

The fact that the logical values are `0` and `1` allows the use of arithmetic operations to perform logical operations. For example, given a vector `x` of numbers, we can check that all the numbers are negative by writing:

```
APL> * fold x < 0
```

For another example, here is an expression that counts the elements that are larger than one given number, e.g., `5`:

```
APL> + fold x > 5
```

Finally, how many numbers are in `x`?[3]

```
APL> + fold x = x
```

[3]For this the latter case, we can also use the pre-defined operation `tally`.

2.6 Restructuring Operations

Let us consider two of the most useful restructuring functions, ι and ρ. For simplicity, we will call them `interval` and `reshape`.

```
APL> interval 7
1 2 3 4 5 6 7
APL> 2 3 reshape 1 2 3 4
1 2 3
4 1 2
APL> 2 3 reshape interval 4
1 2 3
4 1 2
APL> 2 2 reshape 1
1 1
1 1
APL> 2 2 2 reshape 1 2 3 4 5
1 2
3 4

5 1
2 3
```

The monadic function `interval` accepts an integer and produces a vector with all integers from `1` up to that scalar (inclusive). The dyadic function `reshape` creates a tensor with dimensions given in the first argument (the left vector) and fills it with elements of the second argument (the right vector), repeating them as often as necessary to fill the tensor.

Note, in the last example, that the printed form of a three-dimensional tensor is through a sequence of two-dimensional arrays (called tensor *planes*). Each plane is separate from the plane following it by a number of empty lines equal to the number of tensor dimensions minus one.

Another of the most used restructuring functions removes the first n (or last, in case n is negative) components of a tensor: `drop`. Here are some examples:

```
APL> 2 drop interval 10
3 4 5 6 7 8 9 10
APL> -2 drop interval 10
1 2 3 4 5 6 7 8
```

Note that the operator `drop` can remove elements from the various dimensions of a tensor through the use of a vector as first argument:

```
APL> 1 drop 2 3 reshape interval 4
4 1 2
APL> 1 1 drop 2 3 reshape interval 4
1 2
APL> -1 drop 2 2 2 reshape 1 2 3 4 5
1 2
3 4
```

In fact, when the first argument of the function `drop` is a scalar, it is treated as a vector containing only that value.

2.7 Other Operations

Finally, we present two of the most useful operations: the `inner-product` and the `outer-product`.

The operator `inner-product` takes two functions as arguments and produces a third function that, given two matrices, calculates the inner product of the two matrices but instead of employing the traditional algebraic sum of the products of elements, it uses the provided functions. Here are a few examples:

```
APL> m1 =: 2 2 reshape 10 20 30 40
10 20
30 40
APL> m2 =: 2 3 reshape 1 2 3 4 5 6
1 2 3
4 5 6
APL> m1 + inner-product * m2
90 120 150
190 260 330
APL> m1 * inner-product + m2
264 300 338
1364 1440 1518
APL> m1 + inner-product + m2
35 37 39
75 77 79
```

The operator `outer-product` takes a function as argument and produces as a result another function that, given two tensors, produces a third tensor that, for each combination involving one element of each tensors, produces a tensor with the result of applying the function to those two elements. Here is an example:

```
APL> (interval 10) * outer-product interval 10
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

3 Goals

The goal of this project is the implementation, in Common Lisp, of an APL dialect that allows, using the syntax and semantics of Common Lisp, a large set of APL operations.

To that end, you will have to provide:

1. A Common Lisp representation for tensors of numbers and booleans (i.e., the integers `0` and `1`).
2. A set of Common Lisp functions that, operating with the previous representation, implement the corresponging APL operations. The following section details the syntax and semantics of these operations.

Finally, you will have to show the capabilities of your implementation by solving a set of problems specified in a forthcoming section. Your solutions must exclusively use the operations that you implemented and must not use any control structures (particularly, selection, recursion, or iteration).

In the next sections we detail the operations that you need to implement.

3.1 Creation of Scalars and Vectors

The most basic operations that you need to implement allow the creation of scalars and vectors. For ease of use, the operations are named `s`, and `v`.

These operations are used as follows:

```
> (s 1)
1
> (v 1 2 3 4 5 6)
1 2 3 4 5 6
```

As you can see from the previous example, you also need to implement the printing of scalars, vectors and, more generaly, tensors. The rules are as follows:

- If the tensor is a scalar, print its single element.
- If the tensor is a vector, print its elements on the same line separated by one white space.
- If the tensor is a matrix, print its rows as if they were vectors, separated by one empty line.
- If the tensor is not one of the previous cases, then for each sub-tensor of the first dimension, prints the sub-tensor separated from the next sub-tensor by a number of empty lines that is equal to the number of dimensions minus one.

3.2 Monadic Functions

Most of the monadic functions operate element-wise. In order to distinguish them from the pre-defined Common Lisp functions, we will use the same convention adopted in the Julia language: we precede the name with a dot.

`.-` Creates a tensor whose elements are the symmetric of the corresponding elements of the argument tensor.

`./` Same as the previous one, but using the inverse.

`.!` Same as the previous one, but using the factorial.

`.sin` Same as the previous one, but using the sin function.

`.cos` Same as the previous one, but using the cos function.

`.not` Same as the previous one, but using the negation. The result is a tensor containing, as elements, the integers `0` or `1`, depending on the corresponding element in the argument tensor being different that zero or equal to zero.

Here are some examples of the use of the previous functions:[4]

```
> (reshape (v 2 2 2) (v 1 2 3 4 5))
1 2
3 4

5 1
2 3
> (reshape (v 2 2 2 2) (v 1 2 3 4 5))
1 2
3 4

5 1
2 3


4 5
1 2

3 4
5 1
> (.- (.! (v 4 3 2 1 0)))
-24 -6 -2 -1 -1
```

You also need to implement the following functions:

`shape` Creates a vector containing the length of each dimension of the argument tensor.

`interval` Creates a vector containing an enumeration of all integers starting from 1 up to the argument.

Here are a few examples:

```
> (shape (v 1 2 3))
3
> (shape (reshape (v 2 3) (v 1 2 3 4 5 6)))
2 3
> (shape (shape (reshape (v 2 3) (v 1 2 3 4 5 6))))
2
> (interval 6)
1 2 3 4 5 6
```

[4]The `reshape` function has the same semantics as the corresponding APL function and it will be specified later.

```
> (reshape (v 3 3) (interval 6))
1 2 3
4 5 6
1 2 3
```

3.2.1 Dyadic Functions

As was the case of monadic functions, we precede the names of some of the function with a dot to distinguish from the corresponding pre-defined functions in Common Lisp.

`.+` Creates a tensor with the sum of the corresponding elements of the argument tensors. If the arguments are tensors with the same size and shape, the result tensor will have that same size and shape. If one of the arguments is a scalar, the result tensor will have the same size and shape of the other argument and will have, as elements, the sum of the scalar with every element of the other argument. Otherwise, the function signals an error.

`.-` Same as the previous one, but using subtraction.

`.*` Same as the previous one, but using multiplication.

`./` Same as the previous one, but using division.

`.//` Same as the previous one, but using integer division.

`.%` Same as the previous one, but using the remainder of the integer division.

`.<` Same as the previous one, but using the relation "less than." The result tensor will have, as elements, the integers `0` or `1`.

`.>` Same as the previous one, but using the relation "greater than."

`.<=` Same as the previous one, but using the relation "less than or equal to."

`.>=` Same as the previous one, but using the relation "greater than or equal to."

`.=` Same as the previous one, but using the relation "equal to."

`.or` Same as the previous one, but using the logical disjunction.

`.and` Same as the previous one, but using the logical conjunction.

`drop` Accepts a scalar n_1 or vector (of elements n_i) and a non-scalar tensor and returns a tensor where the first (if $n > 0$) or last (if $n < 0$) n elements of the i dimension of the tensor were removed.

`reshape` Returns a tensor with the dimensions refered in the first argument, whose elements are taken from the second argument, repeating them if necessary to fill the resulting tensor.

`catenate` If the two arguments are scalars, returns a vector containing those arguments. If the two arguments are tensors, returns a tensor that joins the arguments along the their last dimension.

`member?` Returns a tensor of booleans with the same shape and dimension of the first argument, containing `1` for each element in the corresponding location in the first argument that occurs somewhere in the second argument and `0` otherwise.

`select` From a tensor of booleans and another tensor, returns a tensor containing only the elements of the last dimension of the second argument whose corresponding element in the first tensor is `1`.

Here are some examples:

```
> (.+ (v 1 2 3) (v 4 5 6))
5 7 9
> (.+ (s 1) (v 4 5 6))
5 6 7
> (.> (s 5) (v 3 2 6 1 4 7 5))
1 1 0 1 1 0 0
> (.< (v 3 1 2) (v 2 3 1))
0 1 0
> (let ((v (v 1 2 3 4 5 6 7 8 9)))
    (.and (.< (s 3) v) (.< v (s 5))))
0 0 0 1 0 0 0 0 0
> (reshape (v 2 2 2) (v 1 2 3))
1 2
3 1

2 3
1 2
> (catenate (v 1 2) (v 3 4 5))
1 2 3 4 5
> (drop (s 2) (interval 10))
3 4 5 6 7 8 9 10
> (drop (s -2) (interval 10))
1 2 3 4 5 6 7 8
> (drop (v 1 1) (reshape (v 3 3) (interval 9)))
5 6
8 9
> (member? (reshape (v 3 3) (interval 4)) (v 1 2))
1 1 0
0 1 1
0 0 1
> (let ((v (v 1 6 2 7 3 0 5 4)))
    (select (.> v (s 3)) v))
6 7 5 4
> (select (v 1 0 1)
          (reshape (v 2 3) (interval 6)))
1 3
4 6
```

3.3 Operators

As was described previously, APL provides a special category of functions known as *operators*. One *operador* is a function that accepts other functions as arguments and returns functions as results.

We will now describe the operators that you need to implement. In order to understand the examples, you need to take into account that Common Lisp has distinct namespaces for functions and variables.

3.3.1 Monadic Operators

`fold` Accepts a function and returns another function that, given a vector, computes the application of the function to sucessive elements of the vector.

`scan` Similar to `fold` but using increasingly large subsets of the elements of the vector, starting from a subset containing just the first element up to a subset containing all elements.

`outer-product` Accepts a function and returns another functions that, given two tensors, returns a new tensor with the result of applying the function to every combination of values from the first and second tensors.

Examples:

```
> (funcall (fold #'.+) (v 1 2 3 4))
10
> (funcall (fold #'.*) (v 1 2 3 4))
24
> (funcall (scan #'.+) (v 1 2 3 4))
1 3 6 10
> (funcall (scan #'.*) (v 1 2 3 4))
1 2 6 24
> (funcall (outer-product #'.=) (interval 4) (interval 4))
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
> (funcall (outer-product #'.=) (v 4 7) (reshape (v 3 4) (interval 12)))
0 0 0 1
0 0 0 0
0 0 0 0

0 0 0 0
0 0 1 0
0 0 0 0
```

3.3.2 Dyadic Operators

`inner-product` Accepts two functions and returns a function that, given two tensors, returns a new tensor computed according to the rules of the algebraic inner product but replacing the algebraic sum and product with the first and second functions.

As an example, consider:

```
> (funcall (inner-product #'.+ #'.*)
    (reshape (v 2 2) (v 10 20 30 40))
    (reshape (v 2 3) (v 1 2 3 4 5 6)))
90 120 150
190 260 330
> (funcall (inner-product #'.* #'.+)
    (reshape (v 2 2) (v 10 20 30 40))
    (reshape (v 2 3) (v 1 2 3 4 5 6)))
264 300 338
1364 1440 1518
```

4 Exercises

Besides the implementation of the requested operations, you need to provide definitions that solve the following exercises. Note that your solutions must only use the functions and operators previously defined and cannot use any control structure.

1. Define the function `tally` that, given a tensor, returns a scalar with the number of elements of the tensor.

   Example:

   ```
   > (tally (reshape (v 3 3 2) (interval 5)))
   18
   > (tally (reshape (v 1 2 3 4) (interval 5)))
   24
   ```

2. Define the function `rank` that, given a tensor, returns a scalar with the number of dimensions of the tensor.

   Example:

   ```
   > (rank (reshape (v 4 5 2) (interval 5)))
   3
   > (rank (reshape (v 4 5) (interval 5)))
   2
   ```

3. Define the function `within` that, given a vector of numbers *v* and two numbers *n1* and *n2*, returns a vector containing only the elements of *v* that are in the range between *n1* and *n2*.

   Example:

   ```
   > (within (v 2 7 3 1 9 8 4 6 5) (s 5) (s 8))
   7 8 6 5
   ```

4. Define the function `ravel` that, given a tensor, returns a vector containing all the elements of the tensor.

   Example:

```
> (ravel (reshape (v 2 3 4) (interval 10)))
1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 10 1 2 3 4
```

5. Define the function `primes` that, given a scalar, returns a vector with all prime numbers from 2 up to the scalar, inclusive.

   Example:

   ```
   > (primes (s 50))
   2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
   ```

4.1 Extensions

You can extend your project to further increase your grade above 20. Note that this increase will not exceed **two** points that will be added to the project grade for the implementation of what was required in the other sections of this specification.

Be careful when implementing extensions, so that extra functionality does not compromise the functionality asked in the previous sections.

Some of the potentially interesting extensions include:

- Allowing the use of numbers without requiring the `s` function.
- Providing a special syntax for vectors that avoids the need of the `v` function.
- Providing additional pre-defined operations.
- Implement a parser and REPL supporting the syntax of APL.

5 Code

Your implementation must work in any Common Lisp implementation, for example, SBCL.

The written code should have the best possible style, should allow easy reading and should not require excessive comments. It is always preferable to have clearer code with few comments than obscure code with lots of comments.

The code should be modular, divided in functionalities with specific and reduced responsibilities. Each module should have a short comment describing its purpose.

We expect all the required functionality to be available in the `COMMON-LISP-USER` package. If you don't define new packages, then you don't need to do anything to fullfil this requirement.

6 Format

Each project must be submitted by electronic means using the Fénix Portal. Each group must submit a single compressed file in ZIP format, named `apl.zip`. Decompressing this ZIP file must generate a folder named `g##`, where `##` is the group's number, containing:

- the source code, within subdirectory `/src`
- the slides of the presentation, in a file named `p2.pdf`
- a `.lisp` file `load.lisp` file that, when loaded, compiles and loads the code into the Common Lisp implementation.

The only accepted format for the presentation slides is PDF. This file must be located at the root of the ZIP file and must have the name `p2.pdf`

7 Evaluation

The evaluation criteria include:

- The quality of the developed solutions.
- The clarity of the developed programs.
- The quality of the public presentation.

In case of doubt, the teacher might request explanations about the inner workings of the developed project, including demonstrations.

The public presentation of the project is a compulsory evaluation moment. Absent students during project presentation will be graded zero in the entire project.

8 Plagiarism

It is considered plagiarism the use of any fragments of programs that were not provided by the teachers. It is not considered plagiarism the use of ideas given by colleagues as long as the proper attribution is provided.

This course has very strict rules regarding what is plagiarism. Any two projects where plagiarism is detected will receive a grade of zero.

These rules should not prevent the normal exchange of ideas between colleagues.

9 Final Notes

Don't forget Murphy's Law.

10 Deadlines

The code and the slides must be submitted via Fénix, no later than 23:00 of **May, 15**.

The presentations will be done during the classes after the deadline. Only one element of the group will present the work and the presentation must not exceed 15 minutes. The element will be chosen by the teacher just before the presentation. Note that the grade assigned to the presentation affects the entire group and not only the person that will be presenting. Note also that content is more important than form. Finally, note that the teacher may question any member of the group before, during, and after the presentation.