CLisp APL Dialect

Advanced Programming

Técnico Lisboa

Group 10

May 2015

CLisp APL Dialect

- Today's Agenda
    - ✓ Goals
    - ✓ Structure
    - ✓ Implementation Details
        - ➡ Functions (Monadic/Dyadic)
        - ➡ Operators (Dyadic)
        - ➡ Exercises
- Future Improvements

CLisp APL Dialect

- Goals

  - APL is an interactive language, strongly orientated to mathematical concerns, offering a large range of operations over arrays (*i.e.*, tensors).

  - Functions can be of 3 types: Niladic, Monadic and Dyadic.

  - Provide a CLisp implementation for APL, using CLisp syntax and semantic.

CLisp APL Dialect

- <u>Structure</u> (1/6)

  - As said before, APL arrays are tensors.
  - Tensors can be divided in scalars, vectors, matrixes and >2D arrays.
  - Implementation-wise, we specified tensors and scalars.

CLisp APL Dialect

- Structure (2/6)

```
(defclass tensor ()
  ((values :initarg :values
           :reader tensor-values)
   (rank   :initarg :rank
          :reader tensor-rank)
   (shape :initarg :shape
           :reader tensor-shape)
   (size :initarg :size
          :reader tensor-size)))
```

1D array

```
(defclass scalar (tensor) ())
```

CLisp APL Dialect

- <u>Structure</u> (3/6)

```
(defun s (arg)

   "Function that constructs a scalar with the given argument."

   (tensor-construct 'scalar (make-array 1 :initial-contents (list arg)) 0 '() 1))


(defun v (&rest args)

   "Function that constructs a vector with the given arguments."

   (let* ((values (if (listp (first args)) (first args) args))

          (s (list-length values))

          (shape (list s)))

        (tensor-construct 'tensor (make-array shape :initial-contents values) 1
shape s)))
```

CLisp APL Dialect

- <u>Structure</u> (4/6)

```
(defun tensor-apply (function &rest tensors)

   "Function that given one or two tensors of the same size, applies the given
function to them."

   (let ((tensors-values (mapcar #'(lambda (n) (tensor-values n)) tensors))

         (result (tensor-copy-simple (first tensors))))

       (apply #'map-into (tensor-values result) function tensors-values)

   result))
```

CLisp APL Dialect

- <u>Structure</u> (5/6)

```
(defgeneric tensor-apply-dyadic (function tensor1 tensor2))

    (defmethod tensor-apply-dyadic (function (tensor1 tensor) (tensor2 tensor))

         (assert (and (eq (tensor-rank tensor1) (tensor-rank tensor2))

                          (equal (tensor-shape tensor1) (tensor-shape tensor2)))

                  (tensor1 tensor2)

                  "ERROR: The given tensors do not have the same rank nor shape")

         (tensor-apply function tensor1 tensor2))


    (defmethod tensor-apply-dyadic (function (tensor1 scalar) (tensor2 tensor))

         (tensor-apply function (reshape (shape tensor2) tensor1) tensor2))


    (defmethod tensor-apply-dyadic (function (tensor1 tensor) (tensor2 scalar))

         (tensor-apply function tensor1 (reshape (shape tensor1) tensor2)))
```

CLisp APL Dialect

- Structure (6/6)

```
(defmethod print-object ((object tensor) stream)
    "Specialization to tensors of the generic function Print-Object.
     Prints the given tensors according to the rules of dimensions, etc."
    (labels (
        (can-print? (shape indexes)
            (if (null indexes)
                t
                (and (eql (- (car shape) 1) (car indexes))
                     (can-print? (cdr shape) (cdr indexes)))))
        (tensor-print (stream object position shape indexes)
            (let ((cur-dimension (list-length shape))
                  (cur-dimension-value (first shape)))
                (if (eq shape nil)
                    (progn
                        (format stream "~a" (aref (tensor-values object) position))
                        (unless (can-print? (last (tensor-shape object)) (last indexes))
                            (format stream " "))
                        (incf position))
                    (progn
                        (dotimes (dim cur-dimension-value)
                            (setf position (tensor-print stream object position (cdr shape) (append indexes (list dim)))))
                        (unless (can-print? (tensor-shape object) indexes)
                            (format stream "~%"))))
            position)))
    (tensor-print stream object 0 (tensor-shape object) '())))
```

CLisp APL Dialect

- <u>Implementation Details - Monadic Functions</u> (1/4)

<u>Monadic Functions:</u>

(**.-** tensor1 &rest tensor2)
(**./** tensor1 &rest tensor2)
(**.!** tensor)
(**.sin** tensor)
(**.cos** tensor)
(**.not** tensor)
(**shape** tensor)
(**interval** value)

Both functions serve
Monadic/Dyadic Functions

CLisp APL Dialect

- Implementation Details - Monadic Functions (2/4)

```
(defgeneric .- (tensor1 &optional tensor2))
    (defmethod .- (tensor1 &optional tensor2)
        "Specialization of the generic function .- that decides what function to call, taking into account
         the number of given arguments."
        (if (eq tensor2 nil)
              (tensor-apply #'- tensor1)
              (tensor-apply-dyadic #'- tensor1 tensor2)))
    (defgeneric ./ (tensor1 &optional tensor2))
    (defmethod ./ (tensor1 &optional tensor2)
        "Specialization of the generic function ./ that decides what function to call, taking into account
         the number of given arguments."
        (if (eq tensor2 nil)
              (tensor-apply #'(lambda (n) (/ 1 n)) tensor1)
              (tensor-apply-dyadic #'/ tensor1 tensor2)))
```

CLisp APL Dialect

- Implementation Details - Monadic Functions (3/4)

```
(defun .! (tensor) (tensor-apply #'! tensor))

(defun .sin (tensor) (tensor-apply #'sin tensor))

(defun .cos (tensor) (tensor-apply #'cos tensor))

(defun .not (tensor) (tensor-apply #'(lambda (n) (if (eq n 0) 1 0))
tensor))

(defun shape (tensor) (v (tensor-shape tensor)))
```

CLisp APL Dialect

- Implementation Details - Monadic Functions (4/4)

```
(defun interval (value)

  "Monadic function that given an integer, returns a new vector
containing all the integer elements from zero up to the integer."

  (iota (s value)))



(defun iota (scalar)

  (let ((iota-lst '()))

      (dotimes (n (aref (tensor-values scalar) 0))

          (setf iota-lst (append iota-lst (list (+ n 1)))))

  (v iota-lst)))
```

CLisp APL Dialect

- Implementation Details - Dyadic Functions (1/12)

Dyadic Functions:

(**.+** tensor1 tensor2)
(**.*** tensor1 tensor2)
(**.//** tensor1 tensor2)
(**.%** tensor1 tensor2)
(**.<** tensor1 tensor2)
(**.>** tensor1 tensor2)
(**.<=** tensor1 tensor2)
(**.>=** tensor1 tensor2)
(**.=** tensor1 tensor2)
(**.or** tensor1 tensor2)
(**.and** tensor1 tensor2)

(more)

(**reshape** tensor1 tensor2)
(**select** tensor1 tensor2)
(**drop** tensor1 tensor2)
(**catenate** tensor1 tensor2)
(**member?** tensor1 tensor2)

CLisp APL Dialect

- Implementation Details - Dyadic Functions (2/12)

```
(defun .+ (tensor1 tensor2)

  "Dyadic function that given two tensors, returns a new tensor
resulting from applying the sum to each of their elements."

  (tensor-apply-dyadic #'+ tensor1 tensor2))



(defun .< (tensor1 tensor2)

  "Dyadic function that given two tensors, returns a new tensor
resulting from applying the less than relation to each of their
elements."

  (tensor-convert-to-int (tensor-apply-dyadic #'< tensor1 tensor2)))
```

CLisp APL Dialect

- <u>Implementation Details - Dyadic Functions</u> (3/12)

```
(defun .or (tensor1 tensor2)

  "Dyadic function that given two tensors, returns a new tensor resulting
from applying the logical disjunction to each of their elements."

  (tensor-convert-to-int (tensor-apply-dyadic #'(lambda (v1 v2) (or v1 v2))
(tensor-convert-to-bool tensor1) (tensor-convert-to-bool tensor2))))
```

```
(defun .and (tensor1 tensor2)

  "Dyadic function that given two tensors, returns a new tensor resulting
from applying the logical conjunction to each of their elements."

  (tensor-convert-to-int (tensor-apply-dyadic #'(lambda (v1 v2) (and v1
v2)) (tensor-convert-to-bool tensor1) (tensor-convert-to-bool tensor2))))
```

CLisp APL Dialect

- <u>Implementation Details - Dyadic Functions</u> (4/12)

```
(defun reshape (tensor1 tensor2)

   "Dyadic function that given two tensors, returns a new tensor whose shape is the one given
in the 1st argument and contents are the ones from the 2nd argument."

   (let* ((shape (array-to-list (tensor-values tensor1)))

          (size-tensor1 (reduce #'* shape))

          (result-tensor (tensor-construct-simple 'tensor (length shape) shape size-tensor1))

          (tensor2-size (tensor-size tensor2)))

        (dotimes (position size-tensor1)

           (setf (aref (tensor-values result-tensor) position)

                 (aref (tensor-values tensor2) (rem position tensor2-size))))

   result-tensor))
```

CLisp APL Dialect

- Implementation Details - Dyadic Functions (5/12)

```
(defun member? (tensor1 tensor2)

  "Dyadic function that given two tensors, returns a new tensor resulting from
testing if each element of 1st tensor is present somewhere on the 2nd tensor."

  (let ((result-tensor (tensor-copy-simple tensor1)))

      (dotimes (position (tensor-size tensor1))

          (if (> (reduce #'+ (tensor-values (.= (s (aref (tensor-values
tensor1) position)) tensor2))) 0)

              (setf (aref (tensor-values result-tensor) position) 1)

              (setf (aref (tensor-values result-tensor) position) 0)))

  result-tensor))
```

CLisp APL Dialect

- <u>Implementation Details - Dyadic Functions</u> (6/12)

(SELECT)

<u>1st step</u>: Calculate new Shape

- Keep rank-1 dimensions and concatenate the sum of ‘1’s in 1st Tensor.

<u>2nd step</u>: Transform tensor (array) into list.

<u>3rd step</u>: Apply recursion (constructs a new list)

- When in last dimension (columns), only add values to list whose position in 1st tensor is 1.

<u>4th step</u>: Transform resulting list into array and create new Tensor with new Shape and new Values.

CLisp APL Dialect

- Implementation Details - Dyadic Functions (7/12)

(SELECT - Example)

t1 **t2**

```
> (select (v 1 0 1) (reshape (v 2 3) (interval 6)))
```

1st step: (original-shape : (2 3)); (dropped-shape : (2)) ; (number-of-1’s : 2) **->** (resulting-shape : (2 2))

2nd step: ((1 2 3) (4 5 6))

3rd step: ((1 2 3) (4 5 6)) -> ((1 0 1) (1 0 1)) -> ((1 3) (4 6))

4th step: ((1 3) (4 6)) -> [1 3 5 6]

CLisp APL Dialect

- <u>Implementation Details - Dyadic Functions</u> (8/12)

```
(defgeneric catenate (tensor1 tensor2))

    (defmethod catenate ((tensor1 scalar) (tensor2 scalar))

         (v (aref (tensor-values tensor1) 0) (aref (tensor-values tensor2) 0)))


    (defmethod catenate ((tensor1 tensor) (tensor2 scalar))

         (catenate tensor1 (reshape (v (append (array-to-list (tensor-values (drop (s -1) (shape
    tensor1)))) (list 1))) tensor2)))


    (defmethod catenate ((tensor1 scalar) (tensor2 tensor))

         (catenate (reshape (v (append (array-to-list (tensor-values (drop (s -1) (shape
    tensor2)))) (list 1))) tensor1) tensor2))

    (defgeneric catenate (tensor1 tensor2))
```

CLisp APL Dialect

- <u>Implementation Details - Dyadic Functions</u> (9/12)

(CATENATE)

<u>1st step</u>: Test rank (difference between the two must be 0 or 1)

<u>2nd step</u>: Know which tensor is smaller, if they do not have the same rank.

<u>3rd step</u>: If there's a smaller tensor, reshape the smaller tensor with the concatenation of its shape and a 1.

<u>4th step</u>: Calculate shape of resulting tensor (concatenation of rank-1 dimensions with the sum of last dimensions of both tensors).

<u>5th step</u>: Convert arrays to lists.

<u>6th step</u>: Apply recursion (constructs a new list)

    -If on last dimension, add to sub-list the corresponding sublist of the tensor which is being concatenated.

<u>7th step</u>: Transform resulting list into array and create new Tensor with new Shape and new Values.

CLisp APL Dialect

- Implementation Details - Dyadic Functions (10/12)

(CATENATE - Example)

```
                t1                    t2
> (catenate (v 1 1) (reshape (v 2 2) (s 2)))
```

1st step: `(rank_t1: 1 ; rank_t2: 2)`

2nd step: `(diff_ranks: 1) -> (smaller_tensor: t1)`

3rd step: `(shape_t1: (2)) -> (modifiedShape_t1: (2 1)) -> (smaller_tensor: (reshape (v 2 1) (v 1 1))`

4th step: `(shape_smaller: (2 1)) ; (shape_t2: (2 2)) -> (resulting_shape: (2 3))`

5th step: `(smaller_tensor: ((1) (1))) ; (tensor2: ((2 2) (2 2)))`

6th step: `(resulting_list: ((1 2 2) (1 2 2)))`

7th step: `[1 2 2 1 2 2]`

CLisp APL Dialect

- <u>Implementation Details - Dyadic Functions</u> (11/12)

(DROP)

<u>1st step</u>: Calculate new Shape

- If shape given in 1st tensor is smaller than 2nd tensor shape, fill the remaining dimensions with 0's. Then, subtract this filled shape to original shape.

<u>2nd step</u>: Transform tensor (array) into list.

<u>3rd step</u>: Apply recursion (constructs a new list)

- Iterate as many times as the number of elements of tensor 1, applying recursion with increasing depth (increases at each iteration).

    - If n-elements > 0, add the rest of the sub-list.

    - If n-elements < 0, add n-times the first elements of the sub-list.

    - If n-elements = 0, maintains the original sub-list.

<u>4th step</u>: Transform resulting list into array and create new Tensor with new Shape and new Values.

CLisp APL Dialect

- Implementation Details - Dyadic Functions (12/12)

```
(DROP-Example)

            t1                    t2
> (drop (s -1) (reshape (v 2 2) (interval 4)))
```

1st step: `(shape_t2: (2 2)) ; (dropShape: (-1 0)) -> (resulting_shape: (1 2))`

2nd step: `((1 2) (3 4))`

3rd step: `(n_iterations: 1) ; (depth: 0) ; (n_elem_to_drop: -1) -> ((1 2))`

4th step: `((1 2)) -> [1 2]`

CLisp APL Dialect

- Implementation Details - Monadic Operators (1/3)

Monadic Operators:

(**fold** function)

(**scan** function)

(**outer-product** function)

CLisp APL Dialect

- <u>Implementation Details - Monadic Operators</u> (2/3)

```
(defun fold (function)

    "Monadic operator that given a function, returns another function that applies the given one to successive elements of a given vector."

    #'(lambda (tensor)

          (reduce function (map 'array #'s (tensor-values tensor)))))

(defun scan (function)

    "Monadic operator that given a function, returns another function that applies the given one to increasingly larger subsets of the elements of the given vector."

    #'(lambda (tensor)

          (let ((result-list '())

                (scalar-tensor (map 'array #'s (tensor-values tensor))))

                (dotimes (position (tensor-size tensor))

                      (setf result-list (append result-list (list (aref (tensor-values (reduce function scalar-tensor :start 0 :end (+ position 1))) 0)))))

          (v result-list))))
```

CLisp APL Dialect

- <u>Implementation Details - Monadic Operators</u> (3/3)

```
(defun outer-product (function)

   "Monadic operator that given a function, returns another function that applies the given one to all combinations of elements of the given tensors."

   #'(lambda (tensor1 tensor2)

        (let ((shape (append (tensor-shape tensor1) (tensor-shape tensor2)))

               (scalar-tensor (tensor-construct 'tensor (map 'array #'s (tensor-values tensor1)) (tensor-rank tensor1) (tensor-shape tensor1) (tensor-size tensor1)))

               (result-list '()))

               (dotimes (position (tensor-size tensor1))

                   (setf result-list (append result-list (array-to-list (tensor-values (apply function (list (aref (tensor-values scalar-tensor) position) tensor2)))))))

        (tensor-construct 'tensor (list-to-array result-list) (list-length shape) shape (reduce #'* shape)))))
```

CLisp APL Dialect

- <u>Implementation Details - Exercises</u> (1/2)

```
(defun tally (tensor)

   "Function that given a tensor, returns a scalar with the number of elements of the given tensor."

   (funcall (fold #'*) (shape tensor)))

(defun rank (tensor)

   "Function that given a tensor, returns a scalar with the number of dimensions of the given tensor."

   (funcall (fold #'+ ) (.>= (shape tensor) (s 0))))

(defun within (tensor scalar1 scalar2)

   "Function that given a tensor and two scalars, returns a vector containing only the elements of the
given tensor that are in the range between scalar1 and scalar2."

   (select (.and (.>= tensor scalar1) (.<= tensor scalar2)) tensor))

(defun ravel (tensor)

   "Function that given a tensor, returns a vector containing all the elements of the given tensor."

   (reshape (tally tensor) tensor))
```

CLisp APL Dialect

- <u>Implementation Details - Exercises</u> (2/2)

```
(defun primes (scalar)
  "Function that given a scalar, returns a vector containing all the prime elements from 2 up to the scalar, inclusive."
  (let ((droppped-vector (drop (s 1) (iota scalar))))
        (select (.not (member? droppped-vector (funcall (outer-product #'.*) droppped-vector droppped-vector))) droppped-vector)))
```

CLisp APL Dialect

- Future Improvements

  - Implement inner-product.
  - Increase performance, by reducing the number of inner-structure conversions.

Thank you!