The Vector Library

Mark Tarver
December 2012

Standard Vectors

In Shen, a vector is either a *standard vector* or a *non-standard vector.* A standard vector is a vector created by the `vector` function which takes a natural number N and creates a vector V of N elements numbering the first element from 1. The zeroth element V[0] is reserved for N itself indicating the size of the vector. For all indices I, where 0 < I <= N, V[I] is taken up by the failure object and is said to be *undefined*. Given a vector of size N, a call to the Ith element where I is a natural number such that 0 < I <= N is said to be *within bounds*. A vector element that is undefined is printed off as ... .

Any attempt to access an undefined element of a standard vector will raise an exception to the top level unless one of the low level absolute vector functions is used (see below). Effectively this means that nothing can be accessed in a standard vector except its size unless it has been already placed there by the operation of a program command.

The smallest standard vector is the *empty vector* created by the call **(vector 0)** and is also written as **<>**.

The functions **vector->** and **<-vector** described below allow the user to destructively assign and retrieve values from a standard vector. **@v** is a polyadic function that allows elements to be combined into a vector

1.1 The Inbuilt Standard Vector Functions

The following six primitive string functions are defined (see Shen document):

limit : (vector A) --> number
Purpose: returns the size of a vector
Examples:

```
(limit (vector 6))
6 : number

(limit (@v 1 2 3 <>))(vector 0)
3 : number
```

v@
Purpose: polyadic function; non-destructively adds its arguments to the final argument which should be a vector
Examples:

```
(@v a b c <>)
<a b c> : (vector symbol)
```

vector : number --> (vector A)
Purpose: creates a vector on N elements; <> is shorthand for (vector 0).
Examples:

```
(vector 6)
<... ... ... ... ... ...> : (vector A)

(vector 0)
<> : (vector A)
```

Vector? : A --> boolean
Purpose: tests for a vector?
Examples:

```
(vector? <>)
true : boolean

(vector 165)
false : Boolean
```

<-vector : (vector A) --> number --> A
Purpose: accesses the nth element of a vector where n > 0.
Examples:

```
(<-vector (@v 1 2 3 <>) 1)
1 : number

(<-vector (@v 1 2 3 <>) 0)
cannot access 0th element of a vector

(<-vector (@v 1 2 3 <>) 10)
AREF: index 10 for #(3 1 2 3) is out of range
```

Vector-> : (vector A) --> number --> A --> (vector A)
Purpose: destructively modifies a vector
Examples:

```
(datatype just-a-test

  ________________
  (value *vector*) : (vector number);)
just-a-test

(set *vector* (@v 1 2 3 <>))
<1 2 3> : (vector number)

(vector-> (value *vector*) 1 0)
<0 2 3> : (vector number)

(value *vector*)
<0 2 3> : (vector number)
```

1.2 Absolute Vectors and Print Vectors

An *absolute vector* is a non-standard vector which is a vector of the underlying platform. There are no conventions on what may be found in a newly created absolute vector and there are no restrictions on accessing any element of such a vector including the zeroth element. None of the absolute vector functions have types.

In Shen, tuples and standard vectors are absolute vectors and under Common Lisp, so are strings.

A print vector is a non-standard vector where the zeroth element is taken up by a function which determines how the vector is printed.

absvector

Purpose: given a natural number N, creates an absolute vector of size N.

Example:

```
(absvector 3)
<[] [] []> \* the exact nature of the contents is implementation dependent *\
```

absvector? : A --> boolean

Purpose: recognises absolute vectors

Example:

```
(absvector? "Mark")
true : boolean

(absvector? <>)
true : boolean

(absvector? 45)
false : boolean
```

<-address

Purpose: correlate of `<-vector` for absolute vectors

Example:

```
(<-address (absvector 3) 2)
[]
```

address->

Purpose: correlate of `vector->` for absolute vectors

```
(address-> (absvector 3) 2 true)
<[] [] true>
```

2. The Library

We say a vector `V` of size `N` is *dense* iff for every `I, 0 < I <= N, V[I]`, `(<-vector V I)` is defined. The vector definitions in the library are designed to work with both dense and non-dense vectors.

vector-== : (vector A) --> (vector B) --> boolean
Input: a vector V1 and a vector V2
Output: true if the dense copies of the two vectors are equal.
Example:

```
(vector-== (@v 1 2 3 (vector 3)) (@v 1 2 3 (vector 6)))
true : boolean
```

vector-any? : (A --> boolean) --> (vector A) --> boolean
Input: A function F and a vector V.
Output: true just when at least one element of V satisfies F (see **vector-every?**).
Example:

```
(vector-any? symbol? (@v 1 2 3 <>))
false : boolean

(vector-any? (> 2) (@v 1 2 3 <>))
true : boolean
```

vector-append : (vector A) --> (vector A) --> (vector A)
Input: Two vectors of the same type.
Output: the result of appending the two vectors
Example:

```
(vector-append (@v 1 2 3 <>) (@v 4 5 6 <>))
<1 2 3 4 5 6> : (vector number)
```

vector-copy : (vector A) --> (vector A)
Input: A vector V.
Output: A copy of the input.
Example:

```
(vector-copy (@v 1 2 3 <>))
<1 2 3> : (vector number)
```

vector-dense : (vector A) --> (vector A)
Input: A vector V.
Output: A dense copy of the input.
Example:

```
(vector-> (vector 6) 1 a)
<a ... ... ... ... ...> : (vector symbol)

(vector-dense (vector-> (vector 6) 1 a))
<a> : (vector symbol)
```

vector-every? : (A --> boolean) --> (vector A) --> boolean
Input: A function F and a vector V.
Output: true just when all elements of V satisfy F (see **vector-any?**).
Example:

```
(vector-every? number? (@v 1 2 3 <>))
true : boolean

(vector-every? (> 2) (@v 1 2 3 <>))
false : boolean
```

vector-extend : (vector A) --> number --> (vector A)
Input: A vector V and a number N.
Output: A copy of the vector with the size of the vector increased by N.
Example:

```
(vector-extend (vector-> (vector 6) 1 a) 3)
<a ... ... ... ... ... ... ... ...> : (vector symbol)
```

vector-index-defined? : (vector A) --> number --> boolean
Input: A vector V and a number N.
Output: true iff the V[N]th element is defined.
Example:

```
(vector-index-defined? (vector 6) 1)
false : boolean

(vector-index-defined? (@ v a <>) 1)
true : boolean
```

vector-index-undefined? : (vector A) --> number --> boolean
Input: A vector V and a number N.
Output: true iff the V[N]th element is undefined.
Example:

```
(vector-index-undefined? (vector 6) 1)
true : boolean

(vector-index-undefined? (@ v a <>) 1)
false : boolean
```

vector->list : (vector A) --> (list A)
Input: A vector V.
Output: A list of the defined elements of the vector in the order of their occurrence.
Example:

```
(vector->list (@v 1 2 3 <>))
[1 2 3] : (list number)
```

list->vector : (list A) --> (vector A)
Input: A list L.
Output: A vector of the elements of L in the order of their occurrence.
Example:

```
(list->vector [1 2 3])
<1 2 3> : (vector number)
```

vector-map : (A --> B) --> (vector A) --> (vector B)
Input: A function F and a vector V.
Output: A vector V' where F is mapped over the elements of V.
Example:

```
(vector-map (+ 1) (@v 1 2 3 <>))
<2 3 4> : (vector number)
```

vector-map! : (A --> A) --> (vector A) --> (vector A)
Input: A function F and a vector V.
Output: A vector V' where F is destructively mapped over the elements of V.
Example:

```
(value *v*)
<1 2 3> : (vector number)

(vector-map (+ 1) (value *v*))
<2 3 4> : (vector number)

(value *v*)
<2 3 4> : (vector number)
```

vector-prefix? : (vector A) --> (vector B) --> boolean
Input: Vectors V1 and V2.
Output: true just when the V1 is a prefix of V2.
Example:

```
(vector-prefix? (@v 1 2 3 <>) (@v 1 2 3 4 <>))
true : boolean
```

vector-suffix? : (vector A) --> (vector B) --> boolean
Input: Vectors V1 and V2.
Output: true just when the V1 is a suffix of V2.
Example:

```
(vector-suffix? (@v 4 5 6 <>) (@v 1 2 3 4 5 6<>))
true : boolean
```

vector->string : (vector A) --> string
Input: A vector V.

Output: A string where every defined element of V occurs in order.

Example:

```
(vector->string (@v 1 2 3 <>))
“1 2 3” : string
```

string->vector : string --> (vector A)
Input: A string S.

Output: A vector of unit strings of which S is composed.

Example:

```
(string->vector “123”)
<”1” “2” “3”> : (vector string)
```

vector->reverse : (vector A) --> (vector A)
Input: A vector V.

Output: A vector with all elements in reverse order.

Example:

```
(vector-reverse (@v 1 2 3 <>))
<3 2 1> : (vector number)
```

file->vectornum : string --> number --> (vector number)
Input: A string naming a file F and a positive integer N.

Output: A vector of size N with all the bytes read from the file. If the file exceeds the limit of the vector then the remainder is dropped.

Example:

```
(file->vectornum “myfile.txt” 6)
<24 56 78 ... ... ...> : (vector number)
```

file->vectorstring : string --> number --> (vector string)
Input: A string naming a file F and a positive integer N.

Output: A vector of size N with all the bytes read as unit strings from the file. If the file exceeds the limit of the vector then the remainder is dropped.

Example:

```
(file->vectornum “hello.txt” 14)
<”h” “e” “l” “l” “o” “ “ “w” “o” “r” “l” “d” ... ... ...> : (vector string)
```