Scheme 48 User's Guide

Richard A. Kelsey

Abstract

Some of Scheme 48's wonderful features.

1 ASCII character encoding

These are in the structure `ascii`.

- `(char->ascii` *char*`)` → *integer*
- `(ascii->char` *integer*`)` → *char*

These are identical to `char->integer` and `integer->char` except that they use the ASCII encoding.

- ascii-limit — integer
- ascii-whitespaces — list of integers

`Ascii-limit` is one more than the largest value that `char->ascii` may return. `Ascii-whitespaces` is a list of the ASCII values of whitespace characters (space, tab, line feed, form feed, and carriage return).

2 Bitwise integer operations

These functions use the two's-complement representation for integers. There is no limit to the number of bits in an integer. They are in the structures `bitwise` and `big-scheme`.

- `(bitwise-and` *integer integer*`)` → *integer*
- `(bitwise-ior` *integer integer*`)` → *integer*
- `(bitwise-xor` *integer integer*`)` → *integer*
- `(bitwise-not` *integer*`)` → *integer*

These perform various logical operations on integers on a bit-by-bit basis. '`ior`' is inclusive OR and '`xor`' is exclusive OR.

- `(arithmetic-shift` *integer bit-count*`)` → *integer*

Shifts the integer by the given bit count, which must be an integer, shifting left for positive counts and right for negative ones. Shifting preserves the integer's sign.

3 Arrays

These are N-dimensional, zero-based arrays and are in the structure `arrays`.

The array interface is derived from one written by Alan Bawden.

- `(make-array` *value dimension*$_0$ `...)` → *array*
- `(array` *dimensions element*$_0$ `...)` → *array*
- `(copy-array` *array*`)` → *array*

`Make-array` makes a new array with the given dimensions, each of which must be a non-negative integer. Every element is initially set to *value*. `Array` Returns a new array with the given dimensions and elements. *Dimensions* must be a list of non-negative integers, The number of elements should be the equal to the product of the dimensions. The elements are stored in row-major order.

```
(make-array 'a 2 3) → {Array 2 3}

(array '(2 3) 'a 'b 'c 'd 'e 'f)
    → {Array 2 3}
```

`Copy-array` returns a copy of *array*. The copy is identical to the *array* but does not share storage with it.

- `(array?` *value*`)` → *boolean*

Returns `#t` if *value* is an array.

- `(array-ref` *array index*$_0$ `...)` → *value*
- `(array-set!` *array value index*$_0$ `...)`
- `(array->vector` *array*`)` → *vector*
- `(array-dimensions` *array*`)` → *list*

`Array-ref` returns the specified array element and `array-set!` replaces the element with *value*.

```
(let ((a (array '(2 3) 'a 'b 'c 'd 'e 'f)))
  (let ((x (array-ref a 0 1)))
    (array-set! a 'g 0 1)
    (list x (array-ref a 0 1))))
    → '(b g)
```

`Array->vector` returns a vector containing the elements of *array* in row-major order. `Array-dimensions` returns the dimensions of the array as a list.

- `(make-shared-array` *array linear-map dimension*$_0$ `...)` → *array*

`Make-shared-array` makes a new array that shares storage with *array* and uses *linear-map* to map indicies to elements. *Linear-map* must accept as many arguments as the number of *dimension*s given and must return a list of non-negative integers that are valid indicies into *array*.

```
(array-ref (make-shared-array a f i0 i1 ...)
           j0 j1 ...)
```

is equivalent to

```
(apply array-ref a (f j0 j1 ...))
```

As an example, the following function makes the transpose of a two-dimensional array:

```
(define (transpose array)
  (let ((dimensions (array-dimensions array)))
    (make-shared-array array
                       (lambda (x y)
         (list y x))
       (cadr dimensions)
       (car dimensions))))

(array->vector
  (transpose
    (array '(2 3) 'a 'b 'c 'd 'e 'f)))
      → '(a d b e c f)
```

4 Records

New types can be constructed using the `define-record-type` macro from the `define-record-types` structure The general syntax is:

(define-record-type *tag type-name*
  (*constructor-name field-tag* ...)
  predicate-name
  (*field-tag accessor-name* [*modifier-name*])
  ...)

This makes the following definitions:

- *type-name* type
- (*constructor-name field-init* ...) → *type-name*
- (*predicate-name value*) → *boolean*
- (*accessor-name type-name*) → *value*
- (*modifier-name type-name value*)

Type-name is the record type itself, and can be used to specify a print method (see below). *Constructor-name* is a constructor that accepts values for the fields whose tags are specified. *Predicate-name* to a predicate that can returns `#t` for elements of the type and `#f` for everything else. The *accessor-name*s retrieve the values of fields, and the *modifier-name*'s update them. The *tag* is used in printing instances of the record type and the field tags are used in the inspector and to match constructor arguments with fields.

- (`define-record-discloser` *type discloser*)

`Define-record-discloser` determines how records of type *type* are printed. *Discloser* should be procedure which takes a single record of type *type* and returns a list whose car is a symbol. The record will be printed as the value returned by *discloser* with curly braces used instead of the usual parenthesis.

For example

```
(define-record-type pare :pare
  (kons x y)
  pare?
  (x kar set-kar!)
  (y kdr))
```

defines `kons` to be a constructor, `kar` and `kdr` to be accessors, `set-kar!` to be a modifier, and `pare?` to be a predicate for a new type of object. The type itself is named `:pare`. `Pare` is a tag used in printing the new objects.

By default, the new objects print as `#Pare`. The print method can be modified using DEFINE-RECORD-DISCLOSER:

```
(define-record-discloser :pare
  (lambda (p) `(pare ,(kar p) ,(kdr p))))
```

will cause the result of `(kons 1 2)` to print as `#{pare 1 2}`.

5 Finite record types

The structure `finite-types` has two macros for defining 'finite' record types. These are record types for which there are a fixed number of instances, which are created when the record type is defined. The syntax for the defining a finite type is:

```
(define-finite-type tag type-name
  (field-tag ...)
  predicate-name
  vector-of-elements-name
  name-accessor
  index-accessor
  (field-tag accessor-name [modifier-name])
  ...
  ((element-name field-value ...)
   ...))
```

This differs from `define-record-type` in the following ways:

- No name is specified for the constructor, but the field arguments to the constructor are listed.
- The *vector-of-elements-name* is added; it will be bound to a vector containing all of the elements of the type. These are constructed by applying the (unnamed) constructor to the initial field values at the end of the form.

- There are names for accessors for two required fields, name and index. These fields are not settable, and are not to be included in the argument list for the constructor.
- The form ends with the names and the initial field values for the elements of the type. The name must be first. The remaining values must match the *field-tag*s in the constructor's argument list.
- *Tag* is bound to a macro that maps *element-name*s to the the corresponding element of the vector. The name lookup is done at macro-expansion time.

```
(define-finite-type color :color
  (red green blue)
  color?
  colors
  color-name
  color-index
  (red   color-red)
  (green color-green)
  (blue  color-blue)
  ((white  255 255 255)
   (black    0   0   0)
   (yellow 255 255   0)
   (maroon 176  48  96)))

(color-name (vector-ref colors 0)) → white
(color-name (color black))         → black
(color-index (color yellow))       → 2
(color-red (color maroon))         → 176
```

Enumerated types are finite types whose only fields are the name and the index. The syntax for defining an enumerated type is:

(define-enumerated-type *tag type-name*
  predicate-name
  vector-of-elements-name
  name-accessor
  index-accessor
  (*element-name* ...))

In the absence of any additional fields, both the constructor argument list and the initial field values are not required.

The above example of a finite type can be pared down to the following enumerated type:

```
(define-enumerated-type color :color
  color?
  colors
  color-name
  color-index
  (white black yellow maroon))

(color-name (vector-ref colors 0)) → white
(color-name (color black))         → black
(color-index (color yellow))       → 2
```

6 Hash tables

These are generic hash tables, and are in the structure `tables`. Strictly speaking they are more maps than tables, as every table has a value for every possible key (for that type of table). All but a finite number of those values are `#f`.

- `(make-table)` → *table*
- `(make-symbol-table)` → *symbol-table*
- `(make-string-table)` → *string-table*
- `(make-integer-table)` → *integer-table*
- `(make-table-maker` *compare-proc hash-proc*`)` → *procedure*
- `(make-table-immutable!` *table*`)`

The first four functions listed make various kinds of tables. `Make-table` returns a table whose keys may be symbols, integer, characters, booleans, or the empty list (these are also the values that may be used in `case` expressions). As with `case`, comparison is done using `eqv?`. The comparison procedures used in symbol, string, and integer tables are `eq?`, `string=?`, and `=`.

`Make-table-maker` takes two procedures as arguments and returns a nullary table-making procedure. *Compare-proc* should be a two-argument equality predicate. *Hash-proc* should be a one argument procedure that takes a key and returns a non-negative integer hash value. If `(`*compare-proc x y*`)` returns true, then `(= (`*hash-proc x*`) (`*hash-proc y*`))` must also return true. For example, `make-integer-table` could be defined as `(make-table-maker = abs)`.

`Make-table-immutable!` prohibits future modification to its argument.

- `(table?` *value*`)` → *boolean*
- `(table-ref` *table key*`)` → *value or #f*
- `(table-set!` *table key value*`)`
- `(table-walk` *procedure table*`)`

`Table?` is the predicate for tables. `Table-ref` and `table-set!` access and modify the value of *key* in *table*. `Table-walk` applies *procedure*, which must accept two arguments, to every associated key and non-`#f` value in `table`.

- `(default-hash-function` *value*`)` → *integer*
- `(string-hash` *string*`)` → *integer*

`default-hash-function` is the hash function used in the tables returned by `make-table`, and `string-hash` it the one used by `make-string-table`.