Fundamentals of Computer Science I: Media Computing (CS151.02 2007F)

Representing Images as Lists of Spots

Summary: We examine Scheme’s *list* data type and an application of that type to representing images.

Contents:

Introduction: Representing Data

As we’ve said a few times, computer scientists study the algorithms we write to manipulate information and the ways in which we represent information. Since an emphasis of this course is images, we have been emphasizing kinds of data related to images. You may not have noticed it, but we have developed a number of ways to represent both colors and images.

For the case of colors, we can represent them as names (e.g., `"blood red"`), as RGB colors (often computed with the `rgb.new`, as in `(rgb.new 102 0 0)`), and as strings that list the three components (e.g., `"102/0/0"`).

For the case of images, we can represent them as a grid of rgb color values that you can get or set. However, as we’ve discovered with our experiments with `region.compute-pixels!`, a function from position to color is sufficient to describe many images.

It’s now time to consider another way to think about images. Particularly for images that have only a few pixels set, we can describe the image by indicating the positions and colors of those pixels. (We can also describe a small part of the image in this way. Such descriptions can then act as a form of stamp for the image.) We call call the combination of position and color a *spot*.

Scheme’s Lists to the Rescue

Fortunately, Scheme, like Lisp before it, provides a simple and elegant way to represent collections of data, the *list*. In contrast to Scheme’s “unstructured” data types, such as symbols and numbers, lists are “structures” that contain other values as elements. In particular, **a list is an ordered collection of values.** In Scheme, lists can be *heterogeneous*, in that they may contain different kinds of values.

As we will see throughout this semester, lists provide a simple, relatively convenient, and elegant mechanism for organizing collections of information.

How does Scheme show you that something is a list? It surrounds the list with parentheses and separates the items with spaces. For example, the following is the list of three color names that we like.

```
("red" "grey" "black")
```

For our problem of representing simple images, we can create a list of “spots”, where each spot consists of a column, row, and color. (How do we represent the combined column, row, and color? With another list.) For example, here is the list that might correspond to “a white spot at position (10,11)” (recall that -1 seems to be the integer that DrFu uses for white).

```
(10 11 -1)
```

Similarly, here is a list that might represent red spots at (10,1), (10,2), and (10,3); grey spots at (11,2), (11,3), and (11,4), and black spots at (12,3), (12,4), and (12,5). Note that black seems to be represented by 255, grey by -1465341697, and red by -16776961.

```
((10 1 -16776961) (10 2 -16776961) (10 3 -16776961)
 (11 2 -1465341697) (11 3 -2139062017) (11 4 -2139062017)
 (12 3 255) (12 4 255) (12 5 255))
```

Of course, to achieve the goal of usefully representing images as lists of spots, we need to know more than what those lists might look like when printed. We must also know basic operations we can use for processing those lists, and others. These include operations for creating lists, for extracting information from lists, for extending lists, and for dealing with each element in a list.

Creating Lists with `list`

The easiest way to create a list is to invoke a procedure named `list`. This procedure takes as many parameters as you care to give it and packs them together into a list.

many of them there may be, and packs them into a list. Just as the addition procedure `+` sums its arguments and returns the result, so the `list` procedure collects its arguments and returns the resulting list:

```
> (list "red" "grey" "black" "white")
("red" "grey" "black" "white")
> (list)
()
> (list 1 2 3 4 5 6 7)
(1 2 3 4 5 6 7)
> (list "red" (rgb.new 255 0 0) (rgb->string (cname->rgb "red")))
```

```
("red" -16776961 "255/0/0")
> (list 10 11 (rgb.new 255 255 255))
(10 11 -1)
> (list 11 2 color.grey)
(11 2 -1465341697)
> (list (list 10 1 color.red)
        (list 10 2 color.red)
        (list 10 3 color.red)
        (list 11 2 color.grey)
        (list 12 3 color.grey))
((10 1 -16776961)
 (10 2 -16776961)
 (10 3 -16776961)
 (11 2 -1465341697)
 (11 3 -2139062017))
```

Nested Lists

You may recall that we said that a list can contain *any* types of Scheme values. As the last two examples above suggest, lists can even contain other lists as values. The technique of placing one list inside another is called *nesting* lists, and is useful in a wide variety of situations.

Most typically, we nest lists when the element lists are used for a different purpose than the enclosing lists. For example, in the examples above, the element lists each represent one spot (that is, column, row, and color) and the list of elements represents a collection of spot.

We are not limited to this simple nesting. We can nest lists within lists within lists within lists, as deep as we desire to go. Not all of our nested values need to be lists. We can, for example, make lists some of whose elements are lists, some of whose elements are strings, some of whose elements are numbers, and so on and so forth.

Constructing Lists with `cons`

Although `list` is a convenient way to build lists, the `list` operation is, itself, composed of other, more primitive, operations. Behind the scenes, `list` invokes `cons` once for each element of the completed list, to hook that element onto the previously created list. The `cons` procedure takes two parameters, a value and a list, and builds a new list by prepending the value onto the list. The prepended value is called the *car* of the resulting list. The previous list (that is, the list that has now had a new value added to the front) is called the *cdr* of the resulting list.

```
> (cons 5 (list 1 2 3))
(5 1 2 3)
```

Why is it called `cons` instead of `list.prepend` or something similar? Well, that’s the name John McCarthy, the designer of Lisp, chose about forty years ago. It’s short for *construct*, because `cons` constructs lists. (The habit of naming procedures with the basic type they operate on, a period, and the key operation did not start until a few decades later.) The names `car` and `cdr` were chosen for very specific reasons that will not make sense for a few more weeks.

Scheme’s name for the empty list is a pair of parentheses with nothing between them: `()`. Most implementations of Scheme suggest permit you to refer to that list as `nil` or `null`. A few permit you to type it verbatim. All permit you to describe the empty list by putting a single quote before the pair of parentheses.

```
> '()
()
> nil
()
> null
()
```

You will find that we prefer to use a name for that list. If sample code does not work in your version of Lisp, try inserting the following definitions.

```
(define nil '())
(define null '())
```

Note that by using `cons` and `nil`, we can build up a list of any length by repeatedly prepending a value to the list.

```
> (define singleton (cons "red" null))
> singleton
("red")
> (define doubleton (cons "green" singleton))
> doubleton
("green" "red")
> (define tripleton (cons "yellow" doubleton))
> tripleton
("yellow" "green" "red")
> (cons "senior" (cons "third-year" (cons "second-year" (cons "freshling" null))))
("senior" "third-year" "second-year" "freshling")
```

You may note that lists built in this way seem a bit “backwards”. That is, the value we add last appears at the front, rather than at the back. However, that’s simply the way `cons` works and, as the last example suggests, in many cases it is a quite sensible thing to do.

Taking Lists Apart

So far, so good. We now know how to build lists using two techniques: Using `list` and using a combination of `cons` and `nil`. Of course, creating lists is not very useful unless we can extract the individual elements of the list.

The two most basic operations one uses to recover elements from a list are `car` and `cdr`. The `car` procedure takes as a parameter a non-empty list and returns the first element. The `cdr` procedure takes as a parameter a non-empty list and returns the remaining elements (that is, all but the first element). In a sense, `car` and `cdr` are the inverses of `cons`; if you think of a non-empty list as having been assembled by a call to the `cons` procedure, `car` gives you back the first argument to `cons` and `cdr` gives you back the second one.

```
> (define colors (cons "red" (cons "black" nil)))
> colors
("red" "black")
> (car colors)
"red"
> (cdr colors)
("black")
```

We can, and often do, nest these calls. For example, to get the second element of a list, we first get the cdr of the list and then take the car of that result.

```
> (car (cdr colors))
"black"
```

To reduce the amount of typing necessary for the programmer, many implementations of Scheme provide procedures that combine `car` and `cdr` in various ways. For example, `cadr` computes the car of the cdr of a list (the second element), `cddr` computes the cdr of the cdr of a list (all but the first two elements), and `caar` computes the car of the car of a list.

```
> (define spots(list (list 10 1 color.red)
                       (list 10 2 color.red)
                       (list 10 3 color.red)
                       (list 11 2 color.grey)
                       (list 12 3 color.grey)))
> spots
((10 1 -16776961) (10 2 -16776961) (10 3 -16776961) (11 2 -1465341697) (11 3 -2139062017))
> (car spots)
(10 1 -16776861)
> (cadr spots)
(10 2 -16776961)
> (caddr spots)
(10 3 -16776961)
> (cadddr spots)
(11 2 -1465341697)
> (caar spots)
10
> (cadar spots)
1
> (caddar spots)
-16776961
```

List Predicates

Scheme provides two basic predicates for checking whether a value is a list. The `null?` predicate checks whether a value is the empty list. The `list?` predicate checks whether a value is a list (empty or nonempty).

```
> (null? nil)
#t
> (list? nil)
#t
> (null? (list 1 2 3))
#f
> (list? (list 1 2 3))
```

```
#t
> (null? 5)
#f
> (list? 5)
#f
```

Other Common List Procedures

It turns out that you can build any other list procedure with just `nil`, `cons`, `car`, `cdr`, and `nil?`. Nonetheless, there are enough common operations that most programmers want to do with lists that Scheme includes them as basic operations. (That means you don't have to define them yourself.) Here are four that programmers frequently use.

`length`

The `length` procedure takes one argument, which must be a list, and computes the number of elements in the list. (An element that happens to be itself a list nevertheless contributes 1 to the total that `length` computes, regardless of how many elements it happens to contain.)

```
> (length nil)
0
> (length (list 1 2 3))
3
> (length (list (list 1 2 3)))
1
```

`reverse`

The `reverse` procedure takes a list and returns a new list containing the same elements, but in the opposite order.

```
> (reverse (list "white" "grey" "black"))
("black" "grey" "white")
```

`append`

The `append` procedure takes any number of arguments, each of which is a list, and returns a new list formed by stringing together all of the elements of the argument lists, in order, to form one long list.

```
> (append (list "red" "green") (list "blue" "yellow"))
("red" "green" "blue" "yellow")
```

The empty list acts as the identify for `append`.

```
> (append nil (list "blue" "yellow"))
("blue" "yellow")
> (append (list "red" "green") nil)
("red" "green")
> (append nil nil)
()
```

`list-ref`

The `list-ref` procedure takes two arguments, the first of which is a list and the second a non-negative integer less than the length of the list. It recovers an element from the list by skipping over the number of initial elements specified by the second argument (applying `cdr` that many times) and extracting the next element (by invoking `car`). So `(list-ref sample 0)` is the same as `(car sample)`, `(list-ref sample 1)` is the same as `(car (cdr sample))`, and so on.

```
> (define rainbow (list "red" "orange" "yellow" "green" "blue" "indigo" "violet"))
> (list-ref rainbow 1)
"orange"
> (list-ref rainbow 0)
"red"
> (list-ref rainbow 5)
"indigo"
> (list-ref rainbow 7)
Error:  --  list-ref "Too few elements in list."
```