CS 61A
Summer 2022

Scheme Data Abstraction

Discussion 10: July 28, 2022 Solutions

Scheme Data Abstractions

Data abstraction is a powerful concept in computer science that allows programmers to treat code as objects. For example, using code to represent cars, chairs, people, and so on. That way, programmers don't have to worry about *how* code is implemented; they just have to know *what* it does.

Data abstraction mimics how we think about the world. If you want to drive a car, you don't need to know how the engine was built or what kind of material the tires are made of to do so. You just have to know how to use the car for driving itself, such as how to turn the wheel or press the gas pedal.

A data abstraction consists of two types of functions:

- **Constructors**: functions that build the abstract data type.
- **Selectors**: functions that retrieve information from the data type.

Programmers design data abstractions to abstract away how information is stored and calculated such that the end user does *not* need to know how constructors and selectors are implemented. The nature of *abstraction* allows whoever uses them to assume that the functions have been written correctly and work as described. Using this idea, developers are able to use a variety of powerful libraries for tasks such as data processing, security, visualization, and more without needing to write the code themselves!

In Python, you primarily worked with data abstractions using Object Oriented Programming, which used Python `Objects` to store the data. Notably, this is not possible in Scheme, which is a functional programming language. Instead, we create and return new structures which represent the current state of the data.

Cities

Say we have an abstract data type for cities. A city has a name, a latitude coordinate, and a longitude coordinate.

Our data abstraction has one **constructor**: * `(make-city name lat lon)`: Creates a city object with the given name, latitude, and longitude.

We also have the following **selectors** in order to get the information for each city:

- `(get-name city)`: Returns the city's name
- `(get-lat city)`: Returns the city's latitude
- `(get-lon city)`: Returns the city's longitude

Here is how we would use the constructor and selectors to create cities and extract their information:

```
scm> (define berkeley (make-city 'Berkeley 122 37))
berkeley
scm> (get-name berkeley)
Berkeley
scm> (get-lat berkeley)
122
scm> (define new-york (make-city 'NYC 74 40))
new-york
scm> (get-lon new-york)
40
```

The point of data abstraction is that we do not need to know how an abstract data type is implemented, but rather just how we can interact with and use the data type.

Q1: Distance

We will now implement the function `distance`, which computes the *Euclidean distance* between two city objects; the Euclidean distance between two coordinate pairs `(x1, y1)` and `(x2, y2)` can be found by calculating the `sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)`. Use the latitude and longitude of a city as its coordinates; you'll need to use the selectors to access this info!

> You may find the following methods useful: - `(expt base exp)`: calculate `base ** exp` - `(sqrt x)` calculate `sqrt(x)`

```
(define (distance city-a city-b)
    (define lat-1 (get-lat city-a))
    (define lon-1 (get-lon city-a))
    (define lat-2 (get-lat city-b))
    (define lon-2 (get-lon city-b))
    (sqrt (+ (expt (- lat-1 lat-2) 2) (expt (- lon-1 lon-2) 2)))
)
```

Q2: Closer City

Next, implement `closer-city`, a function that takes a latitude, longitude, and two cities, and returns the name of the city that is relatively closer to the provided latitude and longitude.

You may only use the selectors and constructors introduced above and the `distance` function you just defined for this question.

> **Hint**: How can you use your `distance` function to find the distance between the given location and each of the given cities?

```
(define (closer-city lat lon city-a city-b)
    (define new-city (make-city 'arb lat lon))
    (define dist1 (distance city-a new-city))
    (define dist2 (distance city-b new-city))
    (if (< dist1 dist2) (get-name city-a) (get-name city-b))
)
```

Trees

Here, we have a data abstraction for trees! Recall that a tree instance consists of a label and a list of other tree instances.

Our data abstraction has one **constructor**: * `(tree label branches)`: Creates a tree object with the given label and list of branches. `branches` must be a scheme list of other tree objects, or `nil` if there are no branches.

We also have the following **selectors** in order to get information about a tree:

- `(label t)`: Returns the label of tree `t`
- `(branches t)`: Returns the branchees of tree `t`

Here is an example of using this tree data abstraction:

```
scm> (define t (tree 5 (list (tree 4 nil) (tree 7 nil))))
t
scm> (label t)
5
scm> (label (car (branches t)))
4
scm> (label (car (cdr (branches t))))
7
```

Note that we are unaware of how this tree data abstraction is really implemented, but we are still able to use it through the given constructor and selectors.

Q3: Is Leaf

Implement the `is-leaf?` procedure for the tree data abstraction. `is-leaf?` takes in a tree `t` and returns `#t` if `t` is a leaf and `#f` otherwise.

Recall that a leaf is a tree without any branches.

```
(define (is-leaf? t)
    (null? (branches t))
)
```

Q4: Sum Nodes

Now, consider trees with integer labels. We are interested in finding the sum of all the labels in a tree.

To accomplish this, it would be useful to have the following helper procedure. Implement `sum-list`, which takes in a list of integers `lst` and returns their sum.

```
(define (sum-list lst)
    (if (null? lst)
        0
        (+ (car lst) (sum-list (cdr lst)))
    )
)
```

Now, implement the procedure `sum-nodes`, which takes in a tree `t` and returns the sum of all of its labels. You may assume `t` only consists of integer labels and that `sum-list` works as designed.

> *Note*: The built-in procedure `map` takes in a one-arg procedure and a list, and it returns a list constructed by calling the procedure on each item in the list.

```
(define (sum-nodes t)
    (define branch-sums (map sum-nodes (branches t)))
    (+ (label t) (sum-list branch-sums))
)
```

Q5: Fun Tree

Implement `fun-tree`, which takes in a one-argument procedure `fun` and a tree `t`. It returns a new tree with the same shape as `t`, but each label is the result of applying `fun` to the corresponding label in `t`.

Hint: You may find the `map` procedure useful.

```
(define (fun-tree fun t)
    (define new-label (fun (label t)))
    (define new-branches (map (lambda (b) (fun-tree fun b)) (
    branches t)))
    (tree new-label new-branches)
)
```