

Introduction

Lab 3 was so much fun we are going to do it again, only this time with EOPL's `define-datatype` constructs.

The `define-datatype` constructs are introduced in Section 2.4 of EOPL and are used extensively throughout the rest of the book. These constructs provide data-structure representations on which you can do pattern matching (via the `cases` construct) and which include something close to "type-checking." You can think of them as a Scheme approximation of Haskell's algebraic data types.

To introduce EOPL data-types, we will use them to re-implement the binary tree data type from Lab 3. Like Lab 3, the following is pretty straightforward stuff. So a complex answer probably means you have gone wrong somewhere.

To start, first grab: <http://www.cis.syr.edu/courses/cis352/code/lab4.ss>.

Binary trees via define-datatype

Recall the BNF specification of binary trees from Lab 3:

$$\langle \text{bin-tree} \rangle ::= () \mid ((\langle \text{integer} \rangle) \langle \text{bin-tree} \rangle \langle \text{bin-tree} \rangle) \quad (1)$$

That is, there are two sorts of binary trees:

1. *empty trees* (which have no data)
2. *branches*, which contain: (a) a *key value* (an integer), (b) a *left subtree*, and (c) a *right subtree*.

Here is a data-type definition of these binary trees (from `lab4.ss`):

```
(define-datatype bin-tree bin-tree?
  (empty-tree)
  (branch
   (key integer?)
   (left bin-tree?)
   (right bin-tree?)))
```

The above creates three new procedures, `bin-tree?`, `empty-tree`, and `branch`, where:

```
(bin-tree? x)
  tests whether x is a bin-tree.
```

```
(empty-tree)
  returns the representation of an empty tree.
(branch k lt rt)
  returns the representation of a bin-tree with key: k, left-subtree: lt, and
  right-subtree: rt, provided each of (integer? k) and (bin-tree? lt)
  and (bin-tree? rt) return #t.
```

Load `lab4.ss` into DrScheme and try evaluation the following:

```
(a) (define t1 (branch 4 (empty-tree) (empty-tree)))
(b) (bin-tree? t1)
(c) (bin-tree? 11)
(d) (branch 'x (empty-tree) (empty-tree))
(e) t1
(f) t0
```

Values of data-type objects look pretty strange when printed out, but they really are not for human eyes.

The first line of any data-type definition is of the form

```
(define-datatype type-name type-pred-name
```

where *type-name* is name of the datatype being defined and *type-pred-name* is the name of the predicate for testing whether a datum is of this type or not. The rest of the define-datatype definition consists of a sequence of clauses of the form

```
(variant-name (field-name1 predicate1) ... (field-namek predicatek) )
```

Each *variant-name* is the name of a variant of the datatype being defined. Each *field-name_i* is the name of the the *i*-th field of the variant being introduced and *predicate_i* is the "of type" *predicate_i*. Note that the notion of "type" here is a little goofy: *predicate_i* is a test that a datum has to pass to be accepted as a *field-name_i*. See, for example, the error raised by `(branch 'x (empty-tree) (empty-tree))` above.

Note that if the variant has no fields (e.g., `empty-tree`), then the clause reduces to just:

```
(variant-name)
```

To compute over datatype, we need to use the `cases` construct. Here is a version of the preorder traversal procedure from Lab 3, rewritten for the `define-datatype` representation binary trees.

```
(define preorder
  (lambda (tree)
    (cases bin-tree tree
      (empty-tree () '())
      (branch (key lt rt)
        (append (list key) (preorder lt) (preorder rt))))))
```

Note that one big advantage of defining data structures this way is that most of the cursed `car`'s and `cdr`'s go away. Try running

```
> (trace preorder)
> (preorder t0)
```

All the `#(struct: somethings` make the trace look even uglier than usual, but the pattern of the recursion is exactly the same as the Lab-3 version of `preorder`.

The first line of a `cases` expression is always of the form

```
(cases type-name expression
```

The `type-name` announces what sort of data-type the `expression` is. Now since the `expression` is of type `type-name`, it has to be one of the variant's of the type. Each, but perhaps the last, clause of the `cases` expression is of the form:

```
(variant-namei (name1 ... namek) resulti)
```

If the value of the `expression` is variant with name `variant-namei`, then:

- ▶ the values of each of the fields of the `expression` are set to `name1` through `namek`,
- ▶ `name1` through `namek` are local variable in the expression `resulti`, and
- ▶ the value `resulti` is be the value of the `cases`-expression in this case.

If the variant has no fields (e.g., `empty-tree`) then the form of the clause is just:

```
(variant-namei () resulti)
```

Also, you can end a `cases` expression with a catch-all clause of the form:

```
(else resultω)
```

If `expression` is not of any of the variants appearing before the else-clause, then the else-clause holds and `resultω` is the result. For example, the following is a predicate for empty bin-trees.

```
(define empty-tree?
  (lambda (x)
    (and (bin-tree? x)
      (cases bin-tree x
        (empty-tree () #t)
        (else #f)))))
```

Let us walk through the development of a `height` procedure for this version of binary trees, where:

```
(height (empty-tree))  ~>  0.
(height k lt rt)       ~>  1 + (height lt) + (height rt)
```

We start with the patently obvious:

```
(define height
  (lambda (tree)
    ???
  ))
```

To classify and take apart the value of tree we need a `cases` expression. So:

```
(define height
  (lambda (tree)
    (cases bin-tree tree
      (empty-tree () ???)
      (branch (k lt rt) ???)
    )))
```

At this point, our specification of `height` tells us how to fill out the rest of the definition:

```
(define height
  (lambda (tree)
    (cases bin-tree tree
      (empty-tree () 0)
      (branch (k lt rt) (+ 1 (height lt) (height rt)))
    )))
```

And that is really all there is to it.

Your Problems

1. Write a Scheme procedure `inorder` that takes a `bin-tree` `tree` and returns the list representing its inorder traversal. *Obvious hint:* Modify `preorder` just like you did in Lab 3.
2. Write a Scheme procedure `max-key` that takes a binary tree with positive integers as key values and returns the maximum key value in the tree. By convention, the maximum of an empty tree is 0.
3. We can represent a list of numbers by

```
(define-datatype lon lon?
  (empty-lon)
  (cell (key number?) (next lon?)))
```

So for example:

```
> (cell 3 (cell 9 (cell -1 (empty-lon))))
#(struct:cell 3
```

```
  #(struct:cell 9
    #(struct:cell -1
      #(struct:empty-lon))))
```

Write two procedures `dt->lst` and `lst->dt` such that

`(dt->lst dt-lon)`

returns the list-representation of `dt-lon`, where `dt-lon` is the datatype representation of a list of numbers. For example:

```
(dt->lst (cell 3 (cell 9 (cell -1 (empty-lon))))
```

should return `(3 9 -1)`.

`(lst->dt lst-lon)`

returns the datatype-representation of `lst-lon`, where `lst-lon` is the standard list representation of a list of numbers. For example:

```
(lst->dt '(3 9 -1))
```

should return the same thing as `(cell 3 (cell 9 (cell -1 (empty-lon))))`.

What to Hand In

- Your source code, including,
 - contracts, and
 - purposes
 for each procedure you are asked to write.
- verified test cases that demonstrate that the code works as required.

You **do not** have to submit anything electronically for this lab.