211lab02_S11

Comp 211 Laboratory 2

Natural Numbers & List Abbreviations

Students should feel free to skip the challenge exercises.

Scheme's Built-in Naturals

We already know Scheme has lots of numbers built-in, like 3, 17.83, and -14/3. It is often convenient to limit our attention to a subset of these such as the naturals: 0, 1, 2, 3, ... We can define the naturals and its template as follows:

```
; A natural (N) is either:
; - 0
; - (add1 n)
; where n is a natural

; Template
; nat-f : natural -> ...
;(define (f ... n ... )
;  (cond [(zero? n) ...]
;        [(positive? n)
;         ... (f ... (sub1 n) ... ) ...]))
```

Of course, we already know what the example data looks like: 0, 1, 2, 3, ...

Unlike most data definitions, we are not defining new Scheme values here (*i.e.*, there's no `define-struct`), but we are defining (identifying) a subset of Scheme numbers. The definition and template use some built-in Scheme functions (`add1`, `sub1`, `zero?`) that may be unfamiliar, but which mean just what their names suggest.

Exercises

Write each of the following functions on **N**.

1. The factorial function `!`, which is defined by the equations:

   ```
   (! 0) = 1
   (! (add1 n)) = (* (add1 n) (! n))
   ```

2. The function `down` that takes an input `n` in **N** and returns the list of **N** `(n ... 1 0)`.
3. The function `up` that takes an input `n` in **N** and returns the list of **N** `(0 1 ... n)`. Hint: define an auxiliary function `upfrom: N N -> list of N` such that `(upfrom m n)` returns `(m (add1 m) ... n)`. Assume that `m` is less than or equal to `n`.

List Abbreviations

Chapter 13 of the book introduces some new, compact methods for representing lists, which have already been mentioned in lecture. The following exercises simply let you explore how this notation behaves.

Finger Exercises on List Abbreviations

1. Evaluate the following in the DrScheme interactions pane. You can cut and paste to save time if you want.

   ```
   (list 1 2 3)
   (cons 1 (cons 2 (cons 3 empty)))
   (list 1 2 3 empty)
   (cons 1 (cons 2 (cons 3 (cons empty empty))))
   ```

2. Rewrite the following using `list`.

```
(cons (cons 1 empty) empty) (cons 1 (cons (cons 2 (cons 3 empty)) (cons 4 (cons (cons 5 empty) empty))))
(cons (cons (cons 'bozo empty) empty) empty)
```

List Constants

Using `'` notation we can abbreviate *constant* lists even more concisely.

Finger Exercises on list constants

1. Evaluate the following in the DrScheme interactions pane. You can cut and paste to save time if you want. Note that `'` produces strange results for embedded references to `true`, `false`, `()`, and `empty`.

```
'(1 2 3 4)
(list 1 2 3 4)
'(rabbit bunny)
(list 'rabbit 'bunny)
'(rabbit (2) (3 4 5))
(list 'rabbit (list 2) (list 3 4 5))
'(true)
'(empty)
'(())
(list empty)
(list ())
(list 'empty)
(list '())
'((cons x y) (1 (+ 1 1) (+ 1 1 1)))
```

Notice that no expressions within the scope of the `'` operator are *evaluated*.

We can think of the `'` operator as distributing over the elements. We apply this rule recursively until there are no more `'` operators left. This simple rule makes embedded references to `true`, `false`, and `empty` behave strangely because `'true`, `'false`, and `'empty` reduce to *themselves* as *symbols*, not to `true`, `false`, and `empty`. In contrast, `'n` for some number `n` reduces to `n`.

Trees and Mutually Recursive Data Definitions

Students should feel free to skip the challenge exercises.

Trees

In class, we used ancestor family trees as an example of inductively defined tree data. In ancestor family trees, each person (a `make-child` structure) has two ancestors (also `make-child` structures) which may be `empty`. In this lab, we'll use a similar, but slightly different, form of tree as an example.

In mathematics, we can use formalized arithmetic expressions as trees. For example,

```
5+(1-8)×(7+1)
```

or equivalently, the Scheme code

```
(+ 5 (* (- 1 8) (+ 7 1)))
```

which encodes expressions as lists revealing their nesting structure.

The string representation for expressions is particularly unattractive for computational purposes because we have to parse the string to understand its structure. The parsing process must understand which symbols are variables, operators incorporate the precedence of infix operators.

We can define a tree formulation of simple Scheme expressions which avoids representing them as list and encodes far more information about their structure. Parsers build tree representations for programs.

To simplify the formulation of Scheme expressions as trees, we will limit each addition, subtraction, multiplication, anddivision operation to exactly two subexpressions. We will limit the atomic elements of expressions to numbers.

```
;; Given

(define-struct add (left right))
(define-struct sub (left right))
(define-struct mul (left right))
(define-struct div (left right))

;; an Arithmetic-Expression (AExp) is either:
;; - a number ;
;; - (make-add l r) where l,r are AExps;
;; - (make-sub l r) where l,r are AExps;
;; - (make-mul l r) where l,r are AExps; or
;; - (make-div l r) where l,r are AExps,

;; Remember that the define-struct function also automatically defines stucture recognizer functions, e.g.
;; add?, sub?, mul? and div?

;; Note: the structure recognizer function, number?, can be used to test if a value is a number.
```

Using this data definition, the arithmetic expression above corresponds to the structure `ae1` defined by

```
(define ae1 (make-add 5 (make-mul (make-sub 1 8) (make-add 7 1))))
```

A trival `AExp` is `ae2` defined by

```
(define ae2 16)
```

Exercises on Arithmetic Expressions

1. Develop the function `eval: AExp -> N` where `(eval ae)` returns the number denoted by the expression `ae`. For example, `(eval ae1)` should return `-51`, and `(eval ae2)` should return `16`.
2. [Challenge] Assume that our expression language includes many basic operations, not just the four supported by `AExp`. We would want a single representation for the application of a binary operator to arguments and use a separate data definition enumerating all of our operations. Rewrite the preceding data definitions, examples, and the function `eval` using for this. As a further challenge, extend your data definition to accommodate unary operations including negation and absolute value as unary operators.

Files and Directories

The following are data definitions are idealized (for the sake of simplicity) representations of files and directories (folders). These definitions follow the Windows convention of attaching a name to a file. They also collapse the definition of the directory type into a clause in the definition of a file, which makes the set f definitions more compact but obfuscates how to write functions that process directories (instead of files). For this reason, none of the following exercises uses a directory as the primary input to a function.

Observe the mutual recursion between files and list-of-files.

```
(define-struct dir (name contents))

; A file is either:
; - a symbol (representing a "simple" file's name) or
; - a directory (make-dir name contents) where name is a symbol, and contents is a lof.


; A list-of-files (lof) is one of
; - empty or
; - (cons f lofd) where f is a file and lofd is a lof
```

This set of definitions is very similar to the descendant trees data structure discussed in class. **Tree-based data structures are very common!**

Directory exercises

1. Create some sample data for the above types.
2. Write the templates for the above types.
3. Develop a function

   ```
   ; find? : symbol file -> boolean
   ; Returns whether the filename is anywhere in the
   ; tree of files represented by the file. This includes both
   ; simple file names and directory names.
   ```

   Note that this function is a vast simplification of{{find}}, the mother-of-all everything-but-the-kitchen-sink UNIX directory traversing command. If open a terminal window and enter

   ```
   man find
   ```

   to see what it can do.

   Use DrScheme's stepper to step through an example use of `find?`. Following the templates leads to an overall strategy known as *depth-first search*, *i.e.*, it explores each tree branch to the end before moving on to the next branch.
4. Develop the following function:

   ```
   ; any-duplicate-names? : file -> boolean
   ; Returns whether any (sub)directory directly or indirectly contains
   ; another directory or file of the same name. It does NOT check
   ; for duplicated names in separate branches of the tree.
   ```

   There is a straightforward way to write this function that just follows the template.
5. Challenge: develop a program to check for duplicated names among all directories and files in the given tree, not just subdirectories.

   Here's a hint. Develop the following function:

   ```
   ; flatten-dir-once : symbol file -> (file or lof)
   ; Purpose: returns a structure like the original file, except that any (sub)directory with that name is
   removed and its contents are promoted up one level in the tree.
   ```

   Here are two pictorial examples, in both cases removing the directory named to-remove. These illustrate why this function can return either a file or a list of files.

Example 1:

```
      foo
    / \   \
 bar baz to-remove
          / \
        one two

becomes

      foo
   /  / \  \
bar baz one two
```

Example 2:

```
 to-remove
   / \ \
foo bar baz

becomes

foo bar baz
```

Follow the templates and think about a single case at a time. If you do that, this exercise is not too difficult. If you don't follow the templates, you are likely to run into difficulty.