Winter 2024
Victoria Sakhnini

Assignment:	7
Due:	Tuesday, March 26, 2024 9:00 pm
Coverage:	End of Module 12
Language level:	Beginning Student with List Abbreviations
Allowed recursion:	Simple, Accumulative and Mutual Recursion
Files to submit:	`components.rkt`, `aexp.rkt`, `filedir.rkt`, `bonus-a07.rkt`

- Make sure you read the A07 Official Post and FAQ post on Piazza for the answers to frequently asked questions.
- Policies from Assignment A06 carry forward.
- For each function you write, you are required to submit the design recipe.

Here are the assignment questions that you need to submit.

1. **(10%):** During manufacturing processes, the final product is often broken down into several smaller sub-components that are combined together in an assembly line. We will represent this idea in a tree:

```
(define-struct component (name num subcomponents))
;; A Component is a (make-component Str Nat (listof Component))
;; Nat is positive
```

The `name` and `num` field represents the component's name and how many of it needs to be manufactured, and the `subcomponents` field represents the list of components needed to assemble together to create it.

For example, a bicycle can be represented as:

```
(define bike (make-component
"bike" 1
(list
 (make-component "frame" 1 empty)
 (make-component "wheel" 2
                 (list
                  (make-component "tire" 1 empty)
                  (make-component "rim" 1 empty)
                  (make-component "spoke" 30 empty)
                  (make-component "hub" 1
                                  (list (make-component "housing" 1 empty)
                                        (make-component "axel" 1 empty)
                                        (make-component "bearing" 20
                                           empty)))))
 (make-component "seat" 1 empty)
 (make-component "handlebar" 1 empty))))
```

Write a function `contains-component?` that consumes a component and a `name` and produces `true` if the component or one of its subcomponents (recursively) has the given name (and `false` otherwise).

For example:

```
(contains-component? bike "hub") => true
(contains-component? bike "brake") => false
```

Place your solution in `components.rkt`.

2. Arithmetic Expressions, Revisited **(40%)**

   At the end of module 10 we discussed binary expression trees and in module 12 we generalized them so that `+` and `*` could consume any number of arguments. In this problem we will generalize them again to include identifiers.

   A Racket expression such as `(+ 2 3 (* 5 10))` is pretty useless. It could easily be replaced by `55`. More useful is something like `(+ x y (* 5 z))` where `x`, `y` and `z` are replaced by the values associated with either parameters or constants.

   Place your solutions in the file `aexp.rkt`.

   (a) On slide M12-04 we had the data definition for Arithmetic Expressions:

   ```
   ;; An Arithmetic Expression (AExp) is one of:
   ;; * Num
   ;; * OpNode

   (define-struct opnode (op args))
   ;; An Operator Node (OpNode) is a
   ;; (make-opnode (anyof '* '+) (listof AExp))
   ```

   Modify this data definition so that arithmetic expressions can include identifiers such as `'x`, `'y`, and `'z`. The earlier example would be represented as

   ```
   (make-opnode '+ (list 'x 'y (make-opnode '* (list 5 'z))))
   ```

   (b) Write the function templates for Arithmetic Expressions based on your modified data definition in Q2a.

   For our marking engine to find your function templates, one of them must be named `aexp-template`. Be sure to check the basic tests for an indication that they have been found.

   (c) Write a new version of `eval` to evaluate expressions that include identifiers. You will need a "symbol table" – a dictionary that associates a symbol with a numerical value. For example,

   ```
   (define a-exp
     (make-opnode '+ (list 'x 'y (make-opnode '* (list 5 'z)))))
   (define sym-table (list (list 'x 1) (list 'y 2) (list 'z 4)))
   (check-expect (eval a-exp sym-table) 23)
   ```

   You may copy the code from M07-35 to look up values in the symbol table. You will need to make one change for it to work with symbols. No documentation is required for `lookup-al`.

   (d) Many programming languages optimize the code before it is executed. One possible optimization is simplifying arithmetic expressions. For example,

- `(make-opnode '+ (list 3 'x 7)` can be replaced with `(make-opnode '+ (list 'x 10))`.
- `(make-opnode '* (list 3 'x 7)` can be replaced with `(make-opnode '* (list 'x 21))`.
- `(make-opnode '* (list 3 'x (make-opnode '+ (list 2 'x 3)) 'z 7))` can be replaced with `(make-opnode '* (list 'x (make-opnode '+ (list 'x 5)) 'z 21))`.
- `(make-opnode '+ (list 2 3 4))` can be replaced with `9`.

In all of these cases, the list of arguments is maintained in the same order except that constants (numbers) are collected and simplified at the end of the list.

Finally, in `(make-opnode '* (list 'x (make-opnode '+ (list 1 2 3)) 3 'z 7))` the `(make-opnode '+ (list 1 2 3))` can be simplified to `(make-opnode '+ (list 6))`. The whole expression can then be simplified to `(make-opnode '* (list 'x 'z 126))`. This is getting hard, and there will be only one or two tests for such cases that are worth only a few marks.

Write `(simplify ex)` which consumes an `AExp` and produces an equivalent `AExp` which has been simplified using the above observations.

This is a hard problem! Some tips:

- Use mutual recursion!
- Some of the techniques for evaluating the expression will be useful.
- One approach (you are free to use others) involves exactly three helper functions:
  i. `(simplify/lst op args acc)` consumes an operator, a list of arithmetic expressions, and an accumulator. Produces the simplified list of arguments. For example:
  ```
  (check-expect (simplify/lst '+ (list 6 'x 7) 0) (list 'x 13))
  ```
  ii. `(simplify/combine op arg1 other-args acc)` consumes an operator, an arithmetic expression, a list of arithmetic expressions, and an accumulator. Produces a new list of arithmetic expressions by adding it to the list if it is a symbol or opnode, otherwise update the accumulator accordingly. For example:
  ```
  (check-expect (simplify/combine '+ 'y (list 'x) 0) (list 'y 'x))
  (check-expect (simplify/combine '+ 5 (list 'x) 0) (list 'x 5))
  ```
  iii. `(maybe-simplify-constant op lst)` consumes an operator and a list of arithmetic expressions. Produces a number or an opcode depending on the list. For example:
  ```
  (check-expect (maybe-simplify-constant '+ (list 9)) 9)
  (check-expect (maybe-simplify-constant  '+ (list 'x 10 'y))
   (make-opnode '+ (list 'x 10 'y)))
  ```

  Yes, the above is cryptic. It leads to a sample solution with about $1\frac{1}{2}$ dozen (18) lines of code, excluding the design recipe. Think really hard about what these functions should produce and then pay strict attention to the contracts.

- Start simple; work up to more complex cases. The order of the above examples is reasonable.

3. **(50%)** For this question, place your solution in the file `filedir.rkt`.

   The provided file `fs-print.rkt` includes the data definitions, some sample data, and some functions to display data. Place the `fs-print.rkt` file in the same directory (folder) as your (saved) `filedir.rkt` file. At the beginning of `filedir.rkt`, include the following:

```
(require "fs-print.rkt")
```

   As a test, try the expression `(fs-print sample-fs)`, which uses the provided print function to display the sample data.

   We will use the following data definitions for representing a file system (file tree) on a computer. **Do not** include the **`define-struct`**s in your file; they will conflict with the ones in `fs-print.rkt`. Alternatively, comment them out.

```
(define-struct file (name size timestamp))
;; A File is a (make-file Str Nat Nat)

(define-struct dir (name contents))
;; A Dir is a (make-dir Str FDList)

;; A FileDir is one of:
;; * a File
;; * a Dir

;; A FDList is one of:
;; * empty
;; * (cons FileDir FDList)
```

   A `FileDir` is a mixed data type that is either a `File` or a `Dir`. A `File` structure includes the name of the file, the size of the file (in bytes) and a timestamp representing the number of seconds since Jan 1, 1970. *(Aside: try entering (current-seconds) in a DrRacket interactions window for such a count.)* A `Dir` structure has a name and a `FDList`, which can be empty.

   Tip: Write templates for a `File`, `Dir`, `FileDir`, and `FDList` and use them as a starting point for your functions.

   (a) Write a function `(list-files fd)` that consumes a `FileDir` and produces a `(listof Str)` that contains the names of all the files (but not the directories) in `fd`. The list should produce the files in the same order as generated by the provided `fs-print` function. That is, the result of `(list-files sample-fs)` will begin with `(list "readme.txt" "zahra.jpg" "timbit.jpg")`. You may use the `append` function.

   Note: to determine the order of `fs-print`, try using the function!

(b) Write a function `backup` that consumes a `Dir` and produces a `Dir` with an extra copy (backup) of each file it contains. Subdirectories also get backed up. The backup files should appear immediately after the original file in the `FDList` that contains this file and have the same size and timestamp, with a ".bak" appended to the name (use `string-append`). That is, the backup of `(make-file "filedir.rkt" 5400 1698197579)` will be

`(make-file "filedir.rkt.bak" 5400 1698197579)`.

(c) Write a function `get-time` that consumes a `FileDir` and produces a timestamp (Nat). For a `File`, it produces the timestamp of the file. For a `Dir`, it produces the latest (largest) timestamp for any file that appears in the `FileSystem` represented by `Dir`, or `false` if there are no files.

(d) Write a function `(find name dir)` that finds every file and directory contained in `dir` that has the given `name`. It produces a list of paths to those files and directories or `empty` if there is no file or directory with the given name. The order of the paths does not matter. `name` is a string. The most common type for `dir` will be a directory but a file should work as well.

```
;; A Path is a (listof Str).
```

Examples:

```
(check-expect (find "vacation" sample-fs)
              (list (list "root" "photos" "vacation")))
(check-expect (find "shopping.txt" sample-fs)
              (list (list "root" "notes" "shopping.txt")))
(check-expect (find "beach1.jpg" sample-fs)
              (list (list "root" "photos" "vacation" "beach1.jpg")))
(check-expect (find "readme.txt"
                    (make-file "readme.txt" 187 1319502441))
              (list (list "readme.txt")))
```

You may use `append` and `reverse`.

This concludes the list of questions for you to submit solutions (but see the following pages as well). Don't forget to always check the basic test results after making a submission.

---

4. **Bonus (5%):**

Implement `(find name dir)` from Q3d again but this time you are not allowed to use append. You may not implement your own version, either. If you met these requirements for Q3d, you may simply resubmit the same code.

Place your solution in `bonus-a07.rkt`.

Enhancements: *Reminder—enhancements are for your interest and are not to be handed in.*

Racket supports unbounded integers; if you wish to compute 2^{10000}, just type `(expt 2 10000)` into the REPL and see what happens. The standard integer data type in most other computer languages can only hold integers up to a certain fixed size. This is based on the fact that, at the hardware level, modern computers manipulate information in 32-bit or 64-bit chunks. If you want to do extended-precision arithmetic in these languages, you have to use a special data type for that purpose, which often involves installing an external library.

You might think that this is of use only to a handful of mathematicians, but in fact computation with large numbers is at the heart of modern cryptography (as you will learn if you take Math 135). Writing such code is also a useful exercise, so let's pretend that Racket cannot handle integers bigger than 100 or so, and use lists of small integers to represent larger integers. This is, after all, basically what we do when we compute by hand: the integer 65,536 is simply a list of five digits (with a comma added just for human readability; we'll ignore that in our representation).

For reasons which will become clear when you start writing functions, we will represent a number by a list of its digits starting from the one's position, or the rightmost digit, and proceeding left. So 65,536 will be represented by the list containing 6, 3, 5, 5, 6, in that order. The empty list will represent 0, and we will enforce the rule that the last item of a list must not be 0 (because we don't generally put leading zeroes on our integers). (You might want to write out a data definition for an extended-precision integer, or EPI, at this point.)

With this representation, and the ability to write Racket functions which process lists, we can create functions that perform extended-precision arithmetic. For a warm-up, try the function `long-add-without-carry`, which consumes two EPIs and produces one EPI representing their sum, but without doing any carrying. The result of adding the lists representing 134 and 25 would be the list representing 159, but the result of the lists representing 134 and 97 would be the list 11, 12, 1, which is what you get when you add the lists 4, 3, 1 and 7, 9. That result is not very useful, which is why you should proceed to write `long-add`, which handles carries properly to get, in this example, the result 1, 3, 2 representing the integer 231. (You can use the warmup function or not, as you wish.)

Then write `long-mult`, which implements the multiplication algorithm that you learned in grade school. You can see that you can proceed as far as you wish. What about subtraction? You need to figure out a representation for negative numbers, and probably rewrite your earlier functions to deal with it. What about integer division, with separate quotient and remainder functions? What about rational numbers? You should probably stop before you start thinking about numbers like 3.141592653589…

Though the basic idea and motivation for this challenge goes back decades, we are indebted to Professor Philip Klein of Brown University for providing the structure here.