Fall 2024
Abouelkhair, Becker, Jung, Petrick, Reetz, Roh

Assignment:	7
Examples Due:	Thursday, November 7, 2024 9:00 pm
Remainder Due:	Tuesday, November 12, 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:	`examples-a07.rkt`, `components.rkt`, `bexp.rkt`, `spelling.rkt`, `bonus-a07.rkt`

General assignment policies

- General policies from previous assignments carry forward. **Policies** from the course web site apply.

A07-specific policies and advice:

- Consult the Official A07 Post on Ed for the answers to frequently asked questions.
- For all functions named in the assignment, examples must be submitted by the "Examples Due"-deadline above.

Here are the assignment questions that you need to submit.

1. **Components (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))
   ;; Requires: num must be larger than 0
   ```

   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 predicate `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:

```
(check-expect (contains-component? bike "hub") true)
(check-expect (contains-component? bike "brake") false)
```

Place your solution in `components.rkt`.

2. **Arithmetic Expressions, Revisited (45%):**

   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 assignment question, we will implement an expression tree that can evaluate Boolean Algebra.

   We want to implement Boolean Algebra from scratch. For that, we need a new data `Boolean`, which can have either the value `0` (for `false`) or `1` (for `true`). **Please be aware that you may NOT use any built-in Boolean functions (e.g., `and`) or built-in Boolean values (e.g., `false`) in your implementation! While you may use `cond`, you are not allowed to nest `cond`-expressions within the same function!**

   (a) Create a new data definition for `Boolean`.

   On slide M12-04 we presented the data definition for an Arithmetic Expression and an Operation Node.

   Based on these, define a `Boolean Expression (BExp)` that must be a `Boolean` (instead of `Num`), or an `OpNode`. Use a fixed-length list instead of a structure in your definition of `OpNode`. Its supported operations include `'AND`, `'OR`, and `'XOR` (instead of `'*` and `'+`).

   In addition, write the function templates based on the data definitions named `BExp` and `OpNode`.

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

(b) Write a new version of `eval` to evaluate Boolean Expressions.

As indicated above, the three Boolean operators that you need to implement are **`AND`**, **`OR`**, and `XOR`. You should be familiar with **`AND`** (see slide M03-11) and **`OR`** by now (see slide M03-12); `XOR` is short for "eXclusive OR": `(xor v1, ..., vn) => 1` if an odd number of `v1 ... vn` have the value `1`, and `=> 0` otherwise.

Both your implementations for **`AND`** and **`OR`** must short-circuit (see slide M03-10). In order to test your implementation for this behaviour, we might violate the contract of `eval`.

```
(check-expect (eval (list 'AND (list 0 1 1))) 0)
(check-expect (eval (list 'OR (list 0 1 1))) 1)
(check-expect (eval (list 'XOR (list 0 1 1))) 0)
(check-expect (eval (list 'AND (list 1 (list 'XOR (list 0 1 (list 'OR
   (list 0 0 0)))) 1))) 1)
```

(c) For this sub-question, we are adding identifiers to our `Boolean Expression`, turning them into a `Boolean Id Expression (BIDExp)`. Identifiers are placeholders in an expression that can later be assigned a value, for example, `'z` in `(list 'OR (list 0 0 'z))`. As a result, a `BIDExp` can now also be a `Sym`.

Furthermore, we want to be able to print out a `BIDExp`. To achieve this, write the function `bidexp->string`, which accepts a `BIDExp` as its single parameter and yields its string-representation in in-fix notation. To represent **`or`**, **`and`**, and `xor`, use `#\+`, `#\*`, and `#\.` respectively; to represent a `Boolean`, use `#\t` and `#\f` instead of `1` and `0`. Make sure to add parenthesis (`#\(` and `#\)`)around each operator to disambiguate their order!

```
(check-expect (bidexp->string (list 'AND (list 0 1 1))) "(f*t*t)")
(check-expect (bidexp->string (list 'OR (list 0 1 1))) "(f+t+t)")
(check-expect (bidexp->string (list 'XOR (list 0 1 1))) "(f.t.t)")
(check-expect (bidexp->string (list 'AND (list 1 (list 'XOR (list 0 1
   (list 'OR (list 0 0 0)))) 1))) "(t*(f.t.(f+f+f))*t)")
(check-expect (bidexp->string (list 'AND (list 1 (list 'AND (list 0
   1))))) "(t*(f*t))")
(check-expect (bidexp->string (list 'XOR (list 0 't 'u 1 'w)))
   "(f.'t.'u.t.'w)")
```

To print an identifier, you may use the function `symbol->string`.

(d) Write a new version of `eval` named `eval-id` to evaluate expressions that includes identifiers. For example,

```
(define identifier-table (list (list 'x 1) (list 'y 0)))

(check-expect (eval-id (list 'AND (list 0 'x 1)) identifier-table) 0)
(check-expect (eval-id (list 'OR (list 'x 'y 1)) identifier-table) 1)
(check-expect (eval-id (list 'XOR (list 0 'y 1)) identifier-table) 1)
```

You will need an "identifier table" – a dictionary (or association list) that associates an identifier (`Sym`) with a value (`Boolean`). You may assume that all identifiers that appear in the expression also appear the the identifier table.

You may use the code from slide M08-25 as a starting point to look up values in the identifier table.

Place your solution in `bexp.rkt`.

3. **Spellbound (45%)**

Spell checking is an examination of an apprentice's ability to perform magic by a qualified instructor (wizard, witch, warlock, shaman, etc.) in a safe environment. Spell-checking is the examination of the correct spelling of a word. Despite the heading of this assignment question, we will, unfortunately, engage in the latter.

There are multiple ways for a computer to check the spelling of a word. A possible approach would be assembling a list of all acceptable words and then comparing the word in question against the list. If the word in question appears in the list, it is spelt correctly; otherwise, it is not. This approach is somehow naïve, as it requires a large amount of memory to store all the words. For starters, we do not know how many words there are, for example, in the English language: even dictionary.com politely refuses to answer this question. Then, the same word oftentimes comes in multiple variations, such as, number (class –classes), tense (sing –sang –sung), or orthography (spelt –spelled), which increases the size of our word list. Also, some languages allow for compounds, such as, Inuktitut and German (example), which increases the size of the word list exponentially. Bottom line: the list-approach works in principle but has severe limitations, and there is an entire field of research dedicated to find better algorithms.

A possible solution is to store words in a (word-) tree, where each node represents a letter of a word. The word-tree below (left), for example, stores the word "FUN". Each node contains the letter it represents (`char`), a Boolean value indicating if a word can terminate on that node (i.e., letter) (`term?`), and a list of nodes representing the next possible letter(s) (`next`) **in that order**. From the word-tree we can deduct, that the words "F" and "FU" would not be valid because the neither the `F`- nor `U`-node can terminate a word (`term?` is `false`), while "FUN" would be valid because the `N`-node can (`term?` is `true`). The example below (right) shows the word-tree after adding the word "FEW".

The word-tree below contains the words "FAR", "FARE", "FED", "FEW", "FEWER", "FEWEST", "FUN", "HAM", and "HAT". Note that this word-tree now contains a "proper" root-node. Unlike any other node in a word-tree, the root-node does not represent a certain

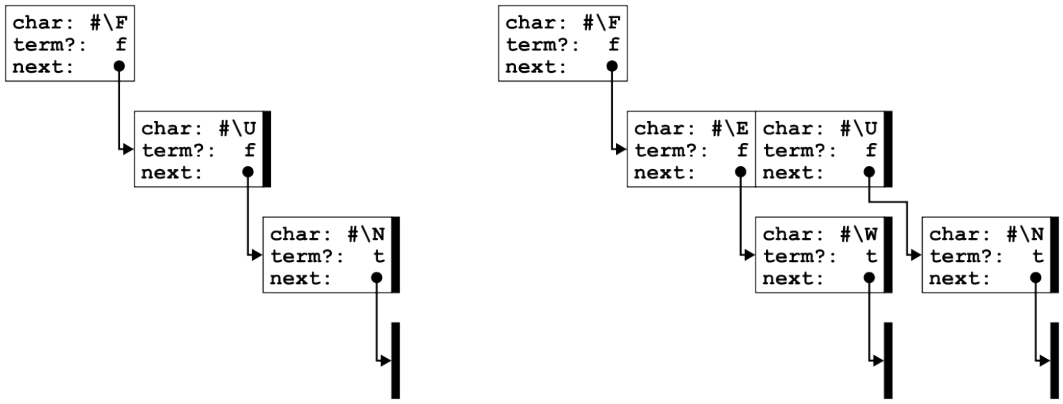

Figure 1: The words "FUN" (left) and "FUN" and "FEW" (right) stored in a tree.

character. Instead, its purpose is merely providing access to the word-tree. We (arbitrarily) decided to store the character `#\space`, but any other character would have worked as well.

In the examples above, we used capital letters only, but your implementation should disregard letter case. This means that the first word-tree not only contains the word "FUN" but also the words "fun", "Fun", "fUn ", "fuN", "fUN", "FuN", and "FUn".

(a) Write the full data definitions for a structure `Node` and a list `Next` as well as the corresponding templates for functions that consumes them. The nodes in `Next` must be sorted in alphabetically ascending order of `char` (i.e., from `A` to `Z`).

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

(b) Write a function `create-tree` that consumes a list of words and produces a `Node` that acts as the root-node for the tree. You may assume that each word is made of lower- and upper-case letters only, i.e, does not contain digits, special characters, or white space.

The `char` stored in the root node must be `#\space`. Again, note that we have omitted a root-node in the Figure 1. In Figure 2 and the `check-expect` below, it is included.

Since letter case does not matter, we expect you to only store upper-case letters as `char` in `Node`.

Examples:

```
(check-expect (create-tree (list "FEW" "FUN"))
              (make-node  #\space false (list
                (make-node  #\F false (list
                  (make-node #\E false (list
                    (make-node #\W true empty)))
                  (make-node  #\U false (list
                    (make-node #\N true empty)))))))))
```

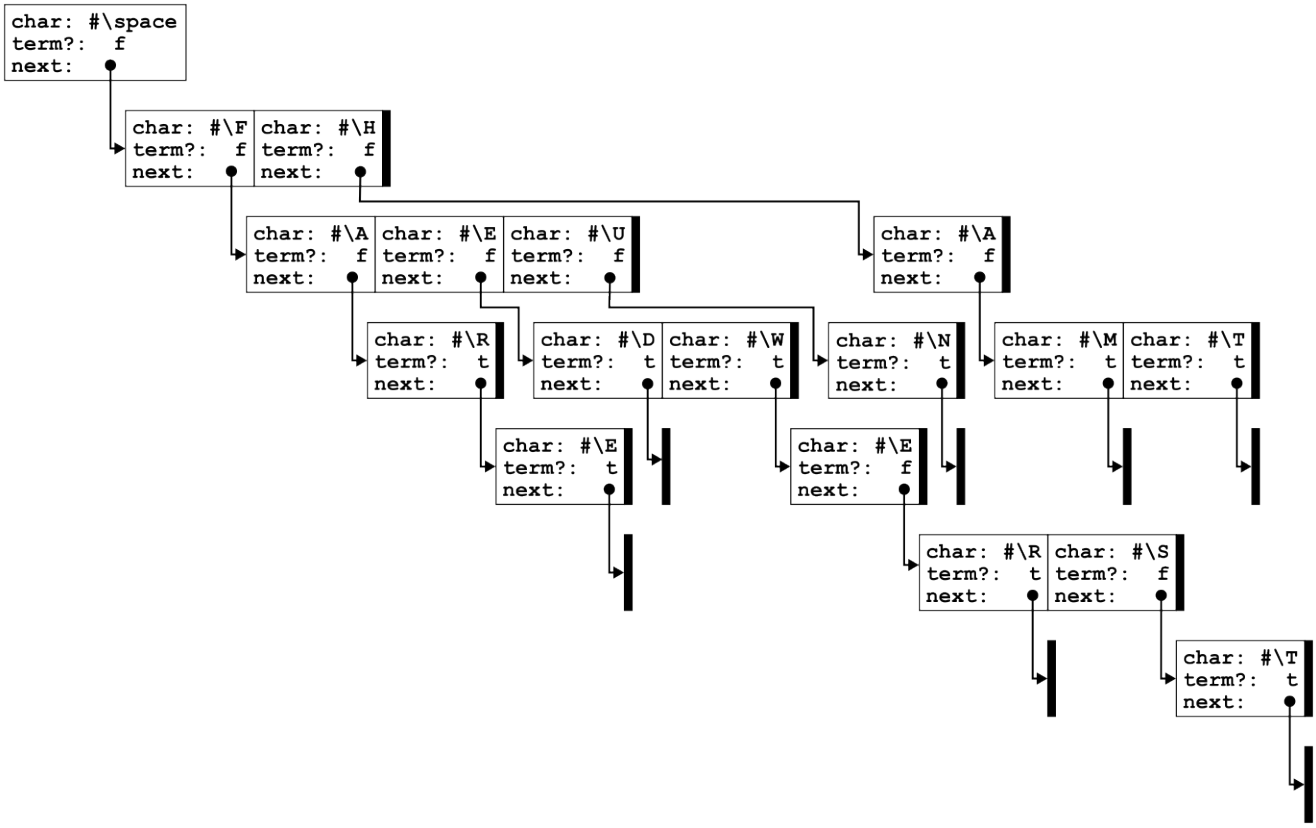

Figure 2: "'FAR", "FARE", "FED", "FEW", "FEWER", "FEWEST", "FUN", "HAM", and "HAT"

```
(check-expect (create-tree (list "FUN" "FAR"))
              (make-node  #\space false (list
                (make-node  #\F false (list
                  (make-node #\A false (list
                    (make-node #\R true empty)))
                  (make-node  #\U false (list
                    (make-node #\N true empty)))))))))

(check-expect (create-tree (list "FUN" "fun" "fUn" "F"))
              (make-node  #\space false (list
                (make-node  #\F true (list
                  (make-node  #\U false (list
                    (make-node #\N true empty)))))))))
```

(c) Write a function `check` that consumes a word and the root-node of a word-tree, and yields `true` if the word is stored in the word-tree, and `false` otherwise.

Examples:

```
(define dict (create-tree (list "FEW" "FUN")))
(check-expect (check "FEW" dict) true)
(check-expect (check "few" dict) true)
(check-expect (check "potatoes" dict) false)
```

Place your solution in `spelling.rkt`.

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

---

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

   Having implemented Boolean Algebra in Q2, we are now able to implement a machine that can add two `Nat` `a` and `b` together. This requires the implementation of three adders. Each of them accepts and yields `Boolean` values.

   First, we need to implement a half-adder. A half-adder has two parameters `A` and `B` and yields the two values `Sum` and `Carry`, as demonstrated by the diagram below. You can think of `Sum` as the sum of `A` and `B`, and `Carry` as an indicator (flag) for a so-called overflow.

   Second, we need to implement a full-adder. A full-adder has three paramemters `A`, `B`, and `Carry-in` and yields the two values `Sum` and `Carry-out`, as demonstrated by the diagram below.

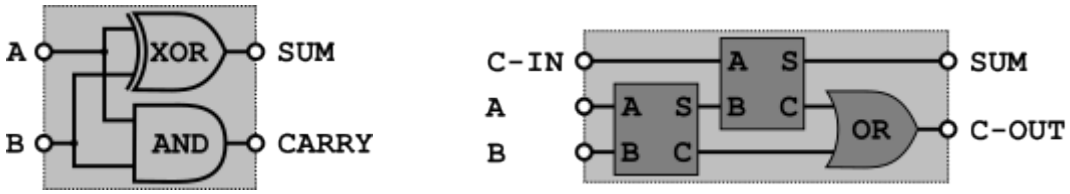

Figure 3: A half-adder built from **`AND`** and `XOR` (left). A full-adder built from two half-adders and an **`OR`** (right).

   Third, we need to implement a ripple-adder. A ripple-adder "wires together" an arbitrary number of full-adders, as demonstrated by the diagram below. Each full-adder is responsible for adding together two bits from two numbers `A` and `B`. The ripple-adder starts by evaluating the least significant bits of `A` and `B`, and uses the `Carry-out` of the previous full-adder as `Carry-in` of the next one. The initial value for `C-I_0` is always `0`, and the final value of `C_O-n` might be used as additional bit in the result.

   See below for an example of how the two numbers `13` (blue) and `5` (pink) are converted into binary, than added together, and than converted back into decimal (green). The Carrt-values are in red.

   Implement the funtion `add`, which yields the sum of two natural numbers `a` and `b`. You may re-use your code from Q2b and/or Q2d. As in Q2, you may not use built-in Boolean functions

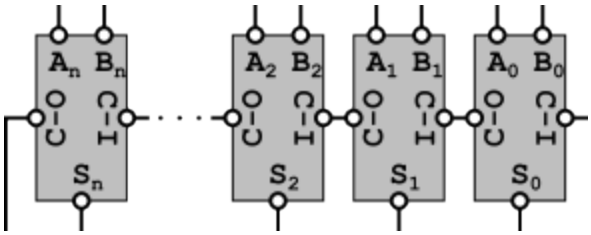

Figure 4: A `n`-bit ripple-adder built from `n` full-adders.

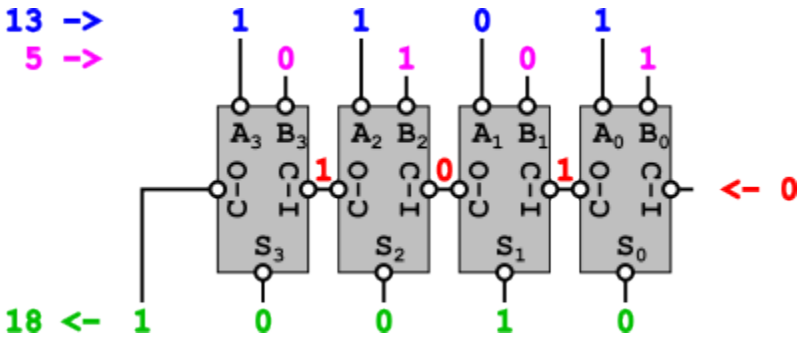

Figure 5: A `4`-bit ripple-adder solving `13 + 5`.

or built-in Boolean values! You also, quite obviously, may not use any built-in arithmetic function, such as, `+` or `*` when implementing half-, full-, and ripple-adder.

```
;; (add a b) calculates the sum of a and b.
;; Example:
(check-expect (add 13 5) 18)
(check-expect (add 135135135 424242) 135559377)

;; add: Nat Nat -> Nat
(define (add a b)
  ...)
```

Hints: Implement two functions `;; dec->bin: Nat -> (listof Boolean)` and `;; bin->dec: (listof Boolean) -> Nat`, which convert a `Nat` between their decimal and binary representation. It is advantageous to have the binary representation reversed, i.e, with the least significant bit at the front of the list. For example, represent `4` as `(list 0 0 1)` instead of `(list 1 0 0)`. For these functions, you may use arithmetic functions, such as, `+` or `remainder`.

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 first 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.