C/C++ 2017/18 exercise 2

This exercise is about representing the syntax of programming languages as C data structures, namely abstract syntax trees, and processing such trees. Learning outcomes: you will use complex data structures in C with struct and union, and recursive functions operating on them.

Here is the grammar of a simple language of expressions (loosely based on Lisp syntax):

E	$\rightarrow$	n	(constant)
E	$\rightarrow$	x	(variable)
E	$\rightarrow$	`(+` L`)`	(operator application for addition)
E	$\rightarrow$	`(*` L`)`	(operator application for multiplication)
E	$\rightarrow$	`(=` x E E`)`	(let binding)
L	$\rightarrow$	E L	(expression list)
L	$\rightarrow$		(empty expression list)

Constants can be integers. Variable names must start with a letter and can be up to 7 characters long. Operator application applies to a list of arguments, e.g.

```
(+ 2 3 5)
```

evaluates to 10. If the list of arguments is empty, the expression evaluates to the neutral element of the operation. For instance, `(+)` evaluates to 0 and `(*)` evaluates to 1.

In a let-binding

```
(= x E1 E2)
```

the variable `x` is bound to the value of E1 in the evaluation of E2. For example,

```
(= x (+ 2 3 5) (* x x (+ x x)))
```

should evaluate to 2000. Note that there could be different scopes for the same variable, so that

```
(= y 2 (+ y (= y 10 y) y))
```

should evaluate to 14 (and not 22).

Evaluation should not change the tree itself. You may assume that there are no variables without a surrounding binding; but if there if your code encoiunters such a variable, you may for example give it the value 0.

In C, the above grammar can be represented by the types in
`http://www.cs.bham.ac.uk/~hxt/2017/c-plus-plus/evalexp.h`

Your task

Write an evaluation function

```
int evalexp(struct exp *e)
```

that evaluates an expression as outlined above. You may also write a helper function that evaluates an expression list.

Given that an expression may contain variables, you will need to write a helper function

```
int evalexpenv(struct exp *e, struct env *env)
```

In addition to an expression, this evaluation function takes an *environment* as a parameter. An environment in this context is a data structure that binds variables to their values. You need to define a suitable data structure. One possibility is a list where each element contains a variable and its corresponding value.

Your submission to Canvas should be a single file evalexp.c implementing the function `evalexp` along with any helper functions and structs, but without a main function. Your code should not print anything.

A sample main file is here:
`http://www.cs.bham.ac.uk/~hxt/2017/c-plus-plus/evalexpmain.c`
Your code should compile with

```
clang -Werror -Wall -o evalexp evalexpmain.c evalexp.c
```

and should of course not produce any compiler or runtime errors.

Marking scheme

2 points are given if your `evalexp` function works on tests without let bindings.

3 more points are given if your `evalexp` function also works on tests with let bindings.

As this exercise is not about memory deallocation, you are not required to deallocate anything. However, code that has memory errors (such as invalid reads or writes) still gets 0 marks. That is, valgrind should not report anything when used like this:

```
valgrind -q ./evalexp
```

Note that there is no flag for leak checking

```
--leak-check=full
```