Project 1: A Spreadsheet Application with DeerLang

In this project, you'll be building the backend for a spreadsheet-like software. We chose this task because it is an example of the type of software that you might build as a software developer. This assignment must be completed using Haskell.

The project is a chance for you to demonstrate that you can

1. Use functional programming to solve a larger problem, where the problem needs to be broken down into parts.
2. Repurpose your interpreter from the previous exercise to a new domain.
3. Use higher-order functions, especially higher-order list functions like `map`, `filter`, and `foldl` to simplify code.
4. Use good programming style while writing functional code.

You may, if you wish, work with one partner to write the code for this project. However, each student must independently write and submit a written description and reflection on the project.

Starter code

The following starter code files can be downloaded from Quercus.

- `Projects/p1/P1.hs`
- `Projects/p1/P1Types.hs`
- `Projects/p1/P1StarterTests.hs`

Submission

There will be two Markus assignments set up:

- "p1": Submit the file `P1.hs`. Do not submit `P1Types.hs`, since we will be supplying our own. Only one person in each team needs to submit this file.
- "p1-writeup": Submit a PDF file `p1-writeup.pdf` containing your written explanation. Each team member must submit this file separately.

If you choose to work with a partner for an assignment, you must form a group for "p1" on MarkUs. One student needs to invite the other to a group. You should declare a partnership well before the deadline (there is no downside of doing so). Ask your instructor for help if you're having trouble forming a group.

Note that "p1-writeup" deadline is set up two days later than "p1", to avoid double-deducting grace tokens. You may submit p1-writeup up 48 hours past the p1 deadline, but we will not accept writeups submitted any later than that (i.e., you cannot use any grace tokens to further extend the writeup deadline).

A spreadsheet application

You've probably used spreadsheet applications like Excel. Not only can you input raw data to Excel, you can also use a formula and have Excel compute a cell value based on values in other cells. In this project, we'll be writing code to compute the values of cells based on its formula written in a language called **DeerLang**.

Our spreadsheet application will not be as fully featured as Excel. Here is an example of the structure of a spreadsheet, with formulas written in DeerLang.

Column:	id	name	age	voter	name2	year-until-100
Formula:				`(canvote age)`	`(concat name name)`	`(- 100 age)`
Values:	1	"adam"	12			
	2	"betty"	15			
	3	"clare"	18			
	4	"eric"	49			
	5	"sam"	17			

Each column has a unique identifier (column name), and can either contain raw data, or a DeerLang formula to be computed. Each row consists of information about one item (in this case, one person). The formula can refer to other columns, and it can refer to both builtin values (like `-`) and user-defined values (like `canvote`) attached to the spreadsheet. For the purposes of this assignment you can assume there will always be at least 1 column with raw data initially and that definitions+identifiers will never be the same.

Here is an example of user-defined values attached to this spreadsheet (in Racket syntax):

```
(define voting-age 18)
(define concat (lambda (x y) (++ x y)))
(define canvote (lambda (x) (>= x voting-age)))
```

Our goal is to fill in the spreadsheet values. In other words, we want to compute the value at each of the blank cells.

Column:	id	name	age	voter	name2	year-until-100
Formula:				`(canvote age)`	`(concat name name)`	`(- 100 age)`
Values:	1	“adam”	12	#f	“adamadam”	88
	2	“betty”	15	#f	“bettybetty”	85
	3	“clare”	18	#t	“clareclare”	82
	4	“eric”	49	#t	“ericeric”	51
	5	“sam”	17	#f	“samsam”	83

In Racket, this example spreadsheet input is represented like this:

```
(define sample-spreadsheet
    '(spreadsheet
       (def (voting-age 18)
            (concat (lambda (x y) (++ x y)))
            (canvote (lambda (x) (>= x voting-age))))
       (columns
         (id    (values 1 2 3 4 5))
         (name  (values "adam" "betty" "clare" "eric" "sam"))
         (age   (values 12 15 18 49 17))
         (voter (computed (canvote age)))
         (name2 (computed (concat name name))
         (year-until-100 (computed (- 100 age)))))))
```

In Haskell, this example spreadsheet input is represented like this:

```
exampleSpreadsheet  =
    Spreadsheet [Def "voting-age" (Literal $ VNum 18),
                 Def "concat" (Lambda ["x", "y"]
                                 (Builtin "++" [Id "x", Id "y"])),
                 Def "canvote" (Lambda ["x"]
                                 (Builtin ">=" [Id "x", Id "voting-age"])) ]
                [ValCol "id" [VNum 1, VNum 2, VNum 3, VNum 4, VNum 5],
                 ValCol "name" [VStr "adam",
                                VStr "betty",
                                VStr "clare",
                                VStr "eric",
                                VStr "sam"],
                 ValCol "age" [VNum 12, VNum 15, VNum 18, VNum 49, VNum 17],
                 ComputedCol "voter" (Apply (Id "canvote") [Id "age"]),
                 ComputedCol "name2" (Apply (Id "concat") [Id "name", Id "name"]),
                 ComputedCol "year-until-100" (Builtin "-" [Literal $ VNum 100,
                                                            Id "age"])]
```

Spreadsheet Specification The Haskell syntax is described by the type definitions in `P1Types.hs`. We'll describe the spreadsheet grammar in Racket here to provide you a second, helpful, way to explore and understand the types. Racket notation has the added advantage of being less verbose.

```
<spreadsheet> = '(' 'spreadsheet'
                  '(' 'def'     (ID <expr>  ) ... ')'
                  '(' 'columns' (ID <column>) ... ')'
                ')'
<column> = '(' 'values' VALUE ... ')'
         | '(' 'computed' <expr> ')'

<expr> = VALUE
       | ID
       | '(' BUILTIN <expr> ... ')'              ; builtin function call
       | '(' 'lambda' '(' ID ... ')' <expr> ')' ; function definition
       | '(' <expr> ... ')'                      ; function expression

VALUE    = ... numbers, strings, booleans ...
BUILTINS = ... described below ...
ID       = ... valid identifiers ...
```

A spreadsheet definition contains two parts: a list of definitions, and a list of columns. We'll describe the structure of each of these in turn.

Definitions The list of definitions is like a `let*` expression from Exercise 3-4, and should be handled similarly. Like a `let*` expression in Racket, there can be zero or more bindings of identifiers to expression values. Just like before, you should evaluate the expressions, and store the resulting values in the environment.

Like in Racket, a later definition in `def` can depend on the value of a previous definition.

```
(def (voting-age 18)
     (canvote (lambda (x) (>= x voting-age)))) ; depend on previous definition
```

A definition **cannot** depend on a later definition:

```
(def (foo (+ 1 voting-age)) ; not allowed!
     (voting-age 18))
```

Note that since the body of a function definition is not evaluated until the function is called, the following definition is okay:

```
(def (voting-age 18)
     (canvote (lambda (x) (>= x voting-age)))) ; ok, because function body not yet evaluated
```

You can assume that definitions **will not be recursive or mutally recursive**: For example, this definition is not allowed.

```
(def (f (lambda (x) (f x)))) ; not allowed: recusive!
```

Neither is this definition:

```
(def (f (lambda (x) (g x)))
     (g (lambda (x) (f x)))) ; not allowed: mutually recursive!
```

Your code does not need to check for these types of invalid definitions; we just won't test with them.

Columns A column can either contain a list of raw values *or* a formula.

You may assume that all value columns have the same number of elements, consistent with the number of rows in the spreadsheet.

A formula column can depend on value columns *or* formula columns, but the columns must be ordered so that later columns only depend on previous columns. Here are some examples of acceptable column ordering:

```
(columns (id (values 1 2 3 4 5))
         (di (values 5 4 3 2 1))) ; only value columns

(columns (id (values 1 2 3 4 5))
         (x  (computed (+ 1 id)))
         (y  (computed (+ 2 x)))) ; computed columns depend on previous

(columns (x  (computed (+ 1 2))) ; computed column can come before value columns
         (id (values 1 2 3 4 5))
         (y  (computed (+ 2 x)))) ; computed columns depend on previous
```

Here are some examples of unacceptable column orders that we will not be testing with:

```
(columns (id (values 1 2 3 4 5))
         (y  (computed (+ 2 x)))   ; NOT valid because the formula depends on later fields
         (x  (computed (+ 1 id))))

(columns (x  (computed (+ 1 x))) ; NOT valid: no recursion!
```

Expressions There are five types of DeerLang expressions:

- A literal value evaluates to itself
- An identifier evaluates to its value in the environment
- A builtin function call evaluates to a call of that builtin function. Notice that some builtins take 2 arguments, and others take a single argument.
- A function definition evaluates to a closure.
- A function expression is evaluated using left-to-right eager evaluation. As before, (1) evaluate the function expression and obtain the closure, (2) evaluate each argument, (3) take the environment from the closure and add to it the mapping of parameter names to the argument values, and (4) evaluate the function body with the new environment from step 3.

Note that unlike in the exercises, a function can now take more than one parameter, or none at all! This could prove to be a bit more challenging. Remember that higher-order list functions like `foldl` are your friends!

Also, since the interpreter can return an `'error` object, we'll need to check and propagate any errors in function/builtin calls.

Errors If there is an error during interpretation, return the value `Error`. In particular, we will test to make sure that you are correctly detecting and reporting the following errors:

1. Unknown identifier: if a referenced identifier does not exist in the environment.
2. Zero division.
3. A function call having too many or too few arguments.
4. A builtin call having too many or too few arguments.
5. A builtin called with the wrong input type (e.g. `(+ 3 "hello")`). (Note: you only need to check the type *dynamically*, when a function is called. A function definition like `(lambda (x) (+ 3 "hello"))` that is not called does not result in an error).

The `Error` Haskell value is of type `Value`. For example, here is the expected output for this spreadsheet application with an error:

Column:	x	y	z
Formula:			`(/ x y)`
Values:	1	0	`Error`
	2	1	2
	3	3	1
	4	2	2
	5	"4"	`Error`

You do not need to detect/prevent other types of errors not on this list (e.g., you only need to check for and detect the five types of errors listed above)

Builtins Our spreadsheet application should support these DeerLang builtin operations. Builtins functions are represented using a string in Haskell (e.g. `Builtin "+" [Id "x", Literal $ VNum 1]`)

Racket Symbol	Racket Function	Haskell String	Haskell Function	Arguments
`'+`	`+`	"+"	`+`	2 numbers
`'-`	`-`	"-"	`-`	2 numbers
`'*`	`*`	"*"	`*`	2 numbers
`'/`	`/`	"/"	`/`	2 numbers
`'>`	`>`	">"	`>`	2 numbers
`'=`	`equal?`	"="	`==`	2 numbers
`'>=`	`>=`	">="	`>=`	2 numbers
`'++`	`string-append`	"++"	`++`	2 strings
`'!`	`not`	"!"	`not`	1 boolean

Output Our main function is called `computeSpreadsheet` in Haskell. It should return a list of columns with computed values. Here's the output structure in Haskell:

```
value-column = ValCol 'col' [value ...]
output = [<value-column> ...]
```

And again for a second helpful representation here's the output structure in Racket:

```
<value-column> = '(' 'values' VALUE ... ')'
<output> = '(' <value-column> ... ')'
```

The list of columns should be in the same order as in the input. Any value columns in the input should also be returned. In other words, if the input spreadsheet has 2 value columns and 3 computed columns, the output should have 5 value columns.

Approaching the Problem

This spreadsheet problem is more involved than the problems that we solved in the exercises. As with any moderately-sized problems, we should spend a significant amount of time planning out our approach (with functional languages, you will often spend more time planning and designing your solution than actually writing code): how can we divide the problem into parts? What helper functions will we need? What are the inputs/outputs to those functions?

We recommend dividing the problem into roughly the following parts. You might want to tackle the problem in various orders.

Writing the DeerLang Interpreter Complete the DeerLang interpreter `evalDeer`. You can use your exercise solutions as a starting point. Write code for the interpreter cases from easiest to hardest. (We provided code to handle the identifiers, since we didn't talk about `Maybe` in Haskell yet.)

Test your interpreter early and often! Come up with as many interesting test cases as you can.

Creating an Environment with the Definitions Write a helper function that builds the environment containing all the definitions. You might want to use your exercise 3 solutions to guide this function.

Write spreadsheet tests To get even more familiar with the spreadsheet problem, write tests that you'll use to test your code. Make sure your tests have good coverage. For example, you might want tests with spreadsheet that has;

- Empty list of definitions, all value columns
- Empty list of definitions, a single computed column that returns a constant
- Empty list of definitions, a single computed column that uses builtins
- Empty list of definitions, a single computed column that uses an anoymous function
- A single function definition, a single computed column that uses that function
- A single function definition, two computed column that uses that function
- ...

The more tests you have, the easier it will be to identify possible issues with your code.

Build Environments with the Data in Each Row Each *row* of your spreadsheet will require its own environment. Write a helper function that takes a list of value columns (only), and returns a list of environments: one for each row.

Evaluate a Single Column Formula, with a Single Environment Write a helper function that takes a single formula, and a list of environment from part 4, and returns a list of values.

Put Everything Together! Combine your helper functions in your main `computeSpreadsheet` function, and test it!

Working with a partner You are encouraged to work with a partner. We expect both partners to contribute equally to the project, and to understand all the code that you submit.

We recommend first scheduling a 1-2 hour meeting to discuss how you want to break down the problem. What are the major helper functions, their inputs/output types? Since you are working in Haskell, you could write the type signatures together. Although these helper functions can change over time, it is a good idea to start with a plan.

You can choose to divide up the helper functions, but you should review each other's code. For the more challenging part of the project, we recommend that you pair-code if possible. In particular:

- Write the `computeSpreadsheet` function together. Or, start by writing an imperfect version of this function together.
- Write the function call portion of the interpreter together.
- Write some of the functions that require `foldl` together.
- Debug together, starting with the simplest test case that fails.

Both partners should write tests.

Style Good Haskell code liberally uses `let` to define local variables to prevent repetition. In fact, expert Haskell code has very little indentation, and a lot of global and local variable bindings!

You should not be afraid to define new helper functions, and to use many, many local variables. For example, it is better to use a local variable called `defs` and assign it to a very simple expression like `(rest (second spreadsheet))`, than to have that expression repeat in many places.

Good Racket and Haskell code is much more succinct than code written in Python, Java, and C/C++. Although it is okay for your code to be more verbose at this point, this project is completable in ~100 lines of code.

Grading

- 20 points: Basic interpreter `evalDeer` that handles identifiers, literal values, builtins, function definition, and function application
- 15 points: Basic evaluation of calculator columns *without* user-defined values; computed columns depend on value columns only
- 15 points: Evaluation of calculator columns *with* user-defined values and functions; computed columns depend on value columns only
- 15 points: Evaluation of calculator columns *with* user-defined values and functions; computed columns can depend on other computed columns

- 15 points: Advanced tests for shadowing, evaluation order
- 10 points: Style
- 10 points: Quality of the Written Explanation

Style Rubric (10 points) We'll be grading for the following:

1. (3 pts) Are you effectively reducing duplicate code? How many code duplications can the TA find in a 5 minute period?
2. (3 pts) Are you documenting your code effectively? Are you choosing good variable names? Haskell names should use `camelCase`, with variable and function names beginning with a lower-case characeter. Can the TA choose a handful of variable names, and understand what the variables store?
3. (4 pts) Can the TA figure out your strategy for handling one particular edge case? We may, for example, ask TAs to see if they can, within 5 minutes, understand how you're generating (say) the environment containing the data for each row.

Written Explanations (10 points) Submit a one-page written (max 500 words, Times New Roman, font size 12) explanation answering the below questions. The explanations should be written and submitted independently, even if you worked with a partner.

1. (4 pt) Describe 3 of your major helper functions (excluding `evalDeer`, since we covered it many times). What are their purposes? Provide a 2-3 sentence summary of how they work, if you don't have that many helper functions you probably have too much repeated code and can break down problems further. If you worked with a partner, choose different functions to write about.
2. (3 pt) Reflect on the use of Haskell. What Haskell language features were helpful in completing this project? What was challenging about using Haskell vs an imperative language?
3. (4 pt) If working in a team, describe your contribution to the project. Otherwise, please declare that you wrote all the code.

The written explantions will be graded based on:

- Can the TA follow your written English with few distractions?
- Are you making correct assertions, and only correct assertions? (Are you wrong about what your code does?)
- Are you demonstrating, clearly, that you have contributed significantly to this project?