Fundies AE2

Given the penguins.csv file, we must analyze several factors about it. Our analyzation must include the following techniques:

- Scalar Processing
- Transformation
- Selection
- Accumulation

Setup

Before we even begin analyzing the data, we must ensure our enviornment is properly set up. Since we are working with tables, we need to have access to table functions. This will cause us to use context dcic2024. Since we are also working with csv files, we need to import two libraries:

- csv
- data-source

This can easily be imported via the keyword "include" followed by the name of the library to be imported.

Importing the table

Since we now have access to the table, we need to import it into the project. This can be done using load-table. Inside the load-table statement, we must include all columns we want to add, in this case, all columns in the csv file:

- index
- species
- island
- bill_length_mm
- bill_depth_mm
- flipper_length_mm
- body_mass_g
- sex
- year

I realized that the first column of the csv file was unnamed. After inspection, it seemed that it only indexed each penguin; thus, I named the column index as it would let me reference it in code.

The final step in importing the table is to sanitize all numeric values. This will let us access them as numbers instead of strings later on. We can do this by writing the following: sanitize "Colname" using num-sanitizer

The process of importing the table should look like the following:

```
penguins = load-table:
  index :: Number,
  species :: String,
```

```
  island :: String,
  bill_length_mm :: Number,
  bill_depth_mm :: Number,
  flipper_length_mm :: Number,
  body_mass_g :: Number,
  sex :: String,
  year :: Number
  source: csv-table-file("penguins.csv", default-options)
  sanitize index using num-sanitizer
  sanitize bill_length_mm using num-sanitizer
  sanitize bill_depth_mm using num-sanitizer
  sanitize flipper_length_mm using num-sanitizer
  sanitize body_mass_g using num-sanitizer
  sanitize year using num-sanitizer
end
```

Scalar Processing

According to Lecture 6.2, Scalar Processing is the following, "Traverse structure and return a single value: sum, count". Here is a problem I came up with relevant to the data that uses Scalar Processing:

Create a function where a user can specify a species and get an average value of one of the columns

Example: Find the average bill length of all the Adelie species

To create this function, I will go through the design process. First, I realized the function will need three parameters, a table, a species, and a column to take the average of. Once I define that, I can begin working on the inner code of the function. The first step working inside the function is to create a temporary table filtered with only penguins of the specified species. This is a trivial task with the provided function filter-with() from the dcic2024 set. The filter with function will require a lambda function that checks if all items in the species column match the species defined by the user. After that is written, we can convert the column specified by the user and convert it to a list via .get-column(). From there, I pass the list into a helper function I created: calcAvg(). We are then able to return the value back to the user. The full function is provided below (There are plenty of comments in the actual file):

```
fun find-Average(table :: Table, species :: String, col :: String) -> Number:
  doc: "Given a table, filter, and column, it will return the average of the
specified column"

  temp-table = filter-with(table, lam(r :: Row): r["species"] == species end)
  avgList = temp-table.get-column(col)
  calcAvg(avgList)
where:
  num-floor(find-Average(penguins, "Adelie", "bill_depth_mm")) is 18
  num-floor(find-Average(penguins, "Chinstrap", "body_mass_g")) is 3733
  num-floor(find-Average(penguins, "Gentoo", "flipper_length_mm")) is 217
end
```

The helper function calcAvg() is very simple. It takes one parameter, a list. As stated before it is a block chain so it has a variable that can change. I created a variable named total. Using a for each loop, I add every value of the list to total. I then take total and divide it by the length to get the average.

```
fun calcAvg(l :: List) -> Number block:
  doc: "Helper function that calculates the average of a given list"

  var total = 0
  for each(n from l):
    total := total + n
  end

  total / l.length()
end
```

Transformation

According to lecture 6.2 Transformation: Produce a new list. Here is a problem I came up with relevant to the data that uses Transformation:

Every measurement is in millimeters, create a function that creates a new column converting a column of a users choice into inches

Example: Convert bill_length_mm into inches

Once again I will think first of the parameters needed for the function. The only two necessary is a table parameter and the name of the column that will be converted (string). My first step is to create the name of the new column. Thankfully the names of the csv file are neat, all ending in their respective measurements. To get the new column name, I simply take the substring of the column defined in the parameter from the first index to the length - 2 (removing mm). I then concatenate in to the end to specify the new measurement. We now have the name of the column so we can work on actually making the column. To create a new column to a table, we can use the function build-column(). This function will use a lambda function inside of it with a row parameter that converts each row into inches.

```
fun toInches(t :: Table, c :: String) -> Table:
  doc: "Converts a column of a table from millimeters to inches"

  newColName = string-substring(c, 0, string-length(c) - 2) + "in"

  build-column(t, newColName, lam(n): num-to-string-digits(n[c] / 25.4, 2) end)

end
```

Selection

According to lecture 6.2 Selection: Select particular elements based on value or position. Here is a problem I came up with relevant to the data that uses Selection:

The user should be able to filter by any island, species, and either greater than or less than any one specific numeric column. Each filter should be toggleable so they can use as many as they want

Example: I want to find any Adelie on the island of torgersen with a bill length under 38

Example 2: I want to find any penguin on the island of Dream

This function requires many parameters as I want users to be able to be as specific as possible. I also want users to not be forced into using any parameter. The parameters needed are the following: A table, a species, an island, a numeric column of the table, a number to compare greater than or less than to, and a boolean value that states if the value should be greater than or less than the given number. Although this seems like its complex, it is mostly built of if else statements. The if statement checks if a field is blank. If it is blank, it will carry on the current table to the next parameter. If the else statement is triggered, it will filter the table according to the parameter. It does this for both the species and island. If it wants to check if a numeric column is greater than a certain amount (ex. bill length > 40) there's a bit more. First we check if the number is 0. If it is we ignore any computation. If not, we check the boolean value. True = greater than, False = less than. It will then use a filter-with and return the final table.

```
fun penguinSearch(t :: Table, species :: String, island :: String, colToCompare ::
String, num :: Number, greatLess :: Boolean) -> Table block:
  doc: "Filters a table with specific inputs given my the user. Leave any string
null to ignore the field and leave the num field as 0 if there is no filter on
numbers"

  caseOne =
    if string-length(species) == 0:
      t
    else:
      filter-with(t, lam(r :: Row): r["species"] == species end)
    end

  caseTwo =
    if string-length(island) == 0:
      caseOne
    else:
      filter-with(caseOne, lam(r :: Row): r["island"] == island end)
    end

  caseThree =
    if num == 0:
      caseTwo
    else:
      if greatLess == true:
        filter-with(caseTwo, lam(r :: Row): r[colToCompare] >= num end)
      else:
        filter-with(caseTwo, lam(r :: Row): r[colToCompare] <= num end)
```

```
      end
    end

  caseThree
end
```

Accumulation

According to lecture 6.2 Accumulation: Pass additional data within the list processing function. Here is a problem I came up with relevant to the data that uses Accumulation:

Create a function that creates a unique identifier for each penguin

Example: ADEN1837M7

This is not that bad of a function to create. It will primarily use the table function build-column alongside some string and number manipulation. I decided the identifier will be the first two letters of the species, last two letters of the island, the first two numbers of the flipper length, the first two numbers of their body mass, the first letter of their sex, and the last number of the year. These can all be accessed using the following functions:

- string-to-upper
- string-substring
- string-length
- num-to-string

```
fun addUID(t :: Table) -> Table:
  doc: "Adds a unique identifyier to all penguins"

  final = build-column(t, "UID", lam(r:: Row): string-to-upper(string-
substring(r["species"], 0, 2)) +  string-to-upper(string-substring(r["island"],
(string-length(r["island"]) - 2), string-length(r["island"]))) + string-
substring(num-to-string(r["flipper_length_mm"]), 0, 2) + string-substring(num-to-
string(r["body_mass_g"]), 0, 2) + string-substring(string-to-upper(r["sex"]), 0,
1) + string-substring(num-to-string(r["year"]), string-length(num-to-
string(r["year"])) - 1, string-length(num-to-string(r["year"]))) end)

  final

end
```