CMPU 101 · Spring 2024

Exam 1 Practice: Solutions

Problem 1

For each of the following Pyret expressions, write what it will evaluate to or, if it will produce an error, write *Error*. Remember to include quotation marks for strings.

a. `42` → `42`

b. `(10 * 4) + 2` → `42`

c. `"4" + 2` → Error

d. `"3" == 3` → `false`

e.
```
n = 42
n == 42
```
→ `true`

f.
```
if 3 > 2 or 2 < 3:
  "Good!"
else:
  "Oh no!"
end
```

→ Error. (Missing parentheses on the first line.)

g.
```
if "Rose" == "rose":
  "Rose is a rose is a rose is a rose."
else if "rose" == "rose":
  "A rose is a rose is a rose"
else:
  "Gertrude Stein is disappointed"
end
```

→ `"A rose is a rose is a rose"`

h. `((5 > 3) or (7 > 2)) and (5 > 2)` → `true`

Problem 2

Consider the following function to return the state of water given a temperature:

```
fun water-state(temp :: Number) -> String:
  doc: "Return a string describing the state of water given its
temperature in degrees Celsius"

  if temp <= 0:
    "solid"
  else if temp < 100:
    "liquid"
  else:
    "gas"
  end

where:


  water-state(-2) is "solid"
  water-state(50) is "liquid"
  water-state(1000) is "gas"


end
```

Fill in the `where` block with enough examples to fully test this function.

Problem 3

Write a function that takes a Number representing a year from 1861 to 1900 and returns a String giving the last name of the person serving as president of Vassar College that year.

Name	*Dates*
Milo P. Jewett	1861–1864
John H. Raymond	1864–1878
Samuel L. Caldwell	1878–1885
James Monroe Taylor	1886–1914

For a year when two people served as president, return the new (that is, later) president's name. See the examples provided in the `where` block below.

This problem should be solved *without* using Pyret tables.

```
fun president-name(year :: Number) -> String:

  doc: "Return the name of the president of Vassar College for a given year"
  if year >= 1915:
     raise("Year is too recent")
  else if year >= 1886:
    "Taylor"
  else if year >= 1878:
    "Caldwell"
  else if year >= 1864:
    "Raymond"
  else if year >= 1861:
    "Jewett"
  else:
    raise("Vassar pre-history")
  end


where:
  president-name(1886) is "Taylor"
  president-name(1885) is "Caldwell"
  president-name(1870) is "Raymond"
  president-name(1861) is "Jewett"
```

```
end
```

Problem 4

You've been hired as a consultant for a major political candidate. The campaign keeps track of donations using a table of the form

```
table: donor :: String, amount :: Number
  ...
end
```

where each value in the amount column is the number of dollars donated, and the donor column has names – or the string "" when the donor is anonymous.

The rules of campaign donation declare that a single contributor may only donate $2,800 to a given campaign. Furthermore, anonymous donations are limited to $50.

a. The first task you are given is to design a function `any-bad-donations` that takes as input a table of donations and returns true if any of the donations in the table are illegal ones.

To make things simpler, you should first design a helper function `is-bad-donation` that answers this question for a single Row.

Before we can design the functions, we need some test data to write examples with:

```
test-table =
  table: donor, amount
    row: "Lynn Burke", 3000
    row: "Robert Wilkins", 50
    row: "", 100
    row: "", 10
  end
```

Then write the helper function:

```
fun is-bad-donation(r :: Row) -> Boolean:
  doc: "Return true if a donation exceeds the appropriate limit for
named or anonymous donations"
  (r["amount"] > 2800)
  or
  ((r["donor"] == "") and (r["amount"] > 50))
where:
```

```
  is-bad-donation(test-table.row-n(0)) is true
  is-bad-donation(test-table.row-n(1)) is false
  is-bad-donation(test-table.row-n(2)) is true
  is-bad-donation(test-table.row-n(3)) is false
end
```

And finally the main function, which uses our helper:

```
fun any-bad-donations(donations :: Table) -> Boolean:
  doc: "Return true if any donation exceeds the applicable limit"
  bad-donations = filter-with(donations, is-bad-donation)
  bad-donations.length() > 0
where:
  any-bad-donations(test-table) is true
  any-bad-donations(table: donor, amount end) is false
end
```

b. Some unscrupulous special-interest groups are attempting to get around the laws on campaign donations by making multiple smaller donations!

   For example, the wealthy but unscrupulous donor Netochka might try to make two donations of $2,800 each:

```
table: donor, amount
  row: "Netochka", 2800
  row: "Netochka", 2800
  ...
end
```

   Design a function `donor-total` that takes as input the name of a donor and a table of donations and produces the total amount donated by the specified person. For example, it would tell us that the total amount donated by `"Netochka"`, given the table of donations above, is $5,600 (which violates the donation limits).

```
test-table2 =
  table: donor, amount
    row: "Netochka", 2800
    row: "Netochka", 2800
    row: "Alice", 50
  end

fun donor-total(donor :: String, donors :: Table) -> Number:
  doc: "Return the sum of all the donations by the given donor in the
given table"

  all-by-donor = filter-with(donors, lam(r): r["donor"] == donor end)

  # This function comes from `dcic-2021`; see the tables documentation
  sum(all-by-donor, "amount")

where:
  donor-total("Alice", test-table2) is 50
  donor-total("Netochka", test-table2) is 5600
end
```

Problem 5

We can define a position as follows:

```
data Posn:
  | posn(x :: Number, y :: Number)
end
```

a. Write an expression that constructs a position with an x-value of 2 and a y-value of 29 and gives it the name `p`.

To create a data value of type `Posn`, use the `posn` constructor that takes in two numbers for its `x` and `y` values:

```
p = posn(2, 29)
```

b. Write an expression that evaluates to the x-value of `p`.

Use the `.` operator in order to access the values of a given data construct:

```
p.x
```

c. Write an expression that constructs a new position with each coordinate of `p` doubled.

We use the `posn` constructor, passing in `2 *` the `x` and `y` coordinates of `p`:

```
posn(2 * p.x, 2 * p.y)
```

Problem 6

Recall the abstract, higher-order functions `filter` and `map`. The following list is used below:

```
NUMS = [list: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

a. Using only the higher-order functions given above, built-in functions, the predicate functions `is-even` and `is-odd` (which you can assume have been defined for you), and lambda expression you write yourself, write expressions for the following:

- Double the elements of `NUMS`.

```
map(lam(n): n * 2 end, NUMS)
```

- Remove all the odd values from `NUMS`.

```
filter(is-even, NUMS)
```

- Remove all the even values from `NUMS`.

```
filter(is-odd, NUMS)
```

b. Evaluate the following expressions:

- `map(num-to-string, NUMS)`

```
[list: "1", "2", "3", "4", "5", "6", "7", "8", "9", "10"]
```

- `filter(lam(x): (x > 5) and (x < 10) end, NUMS)`

```
[list: 6, 7, 8, 9]
```

- `length(filter(lam(x): x == 5 end, NUMS)) > 0`

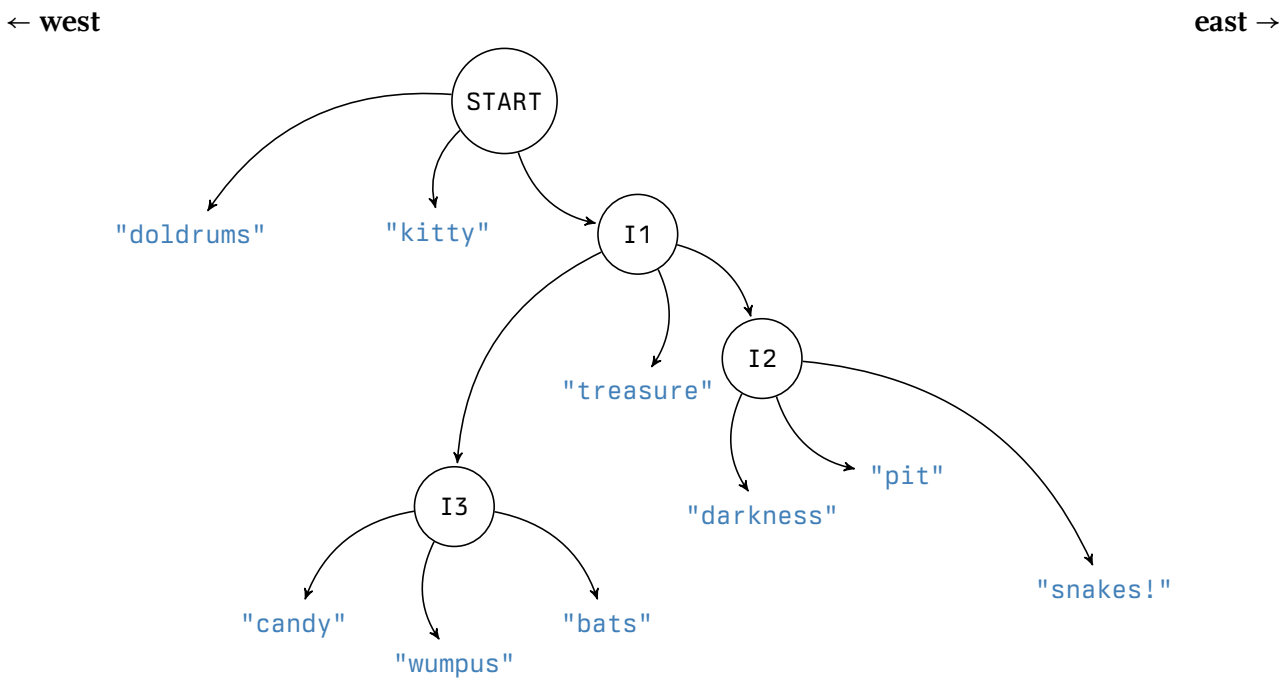
```
true
```

Problem 7

You are in a maze of twisty little passages, all alike. Consider a world made of mazes, which consist of dead ends, represented by a string describing what's there, or intersections that divide into three mazes, continuing to the west, middle, and east:

```
data Maze:
  | dead-end(descr :: String)
  | intersection(west :: Maze, middle :: Maze, east :: Maze)
end
```

a. Define four names to represent this maze:



Hint: Be careful about the *order* of your statements!

```
I3 = intersection(dead-end("candy"), dead-end("wumpus"), dead-end("bats"))
I2 = intersection(dead-end("darkness"), dead-end("pit"), dead-end("snakes!"))
I1 = intersection(I3, dead-end("treasure"), I2)
START = intersection(dead-end("doldrums"), dead-end("kitty"), I1)
```

b. Write a function that takes a Maze as input and returns `true` if it contains the given string:

```
fun maze-contains(maze :: Maze, goal :: String) -> Boolean:
  doc: "Return true if the maze contains the goal"


  cases (Maze) maze:
    | dead-end(d) => d == goal
    | intersection(w, m, e) =>
      maze-contains(w, goal) or
      maze-contains(m, goal) or
      maze-contains(e, goal)
  end


where:
  maze-contains(START, "wumpus") is true
  maze-contains(I3, "pit") is false
  maze-contains(START, "waldo") is false
  maze-contains(I2, "snakes!") is true
end
```