In [ ]:
```
from cs103 import *
```

CPSC 103 - Systematic Program Design

Module 05b Day 2

Adapted by Steve Wolfman from Rik Blok, with thanks to Giulia Toti

---

Ian Mitchell has a personal emergency now. I'll substitute today and Thursday. We're figuring out next week soon!

I'm Steve Wolfman; call me Steve! I've taught CPSC 103 several times, including co-teaching its first offering many years ago 😄. It's been a couple of years, however. So, you may have to help me out on some things!

Reminders

- Wed: Module 4 (Compound): Tutorial Resubmission (optional)
- Wed-Fri: Office hours in tutorials (no attendance taken)
- Thu: Office hours in lieu of class
- Fri: Midterm exam

⚠️ Midterm exam

WHEN: Friday Oct 27th, 2023 @ 6:30PM

WHERE: You have been assigned to a location based on your surname (last/family name).

- IRC 2: Surnames **A-K**
- WESB 100: Surnames **L-R**
- ESB 1013: Surnames **S-Z**
- Students who have registered to write the exam with the CfA will write at the location/time determined by the CfA.
- Students who have submitted an MT Conflict form: please look out for an email sent by the Course Coordinator regarding your exam location and time.

COVERAGE: Modules 1 thru 5 (inclusive).

ID: Bring your UBCcard. Other valid picture ID ok.

NO ELECTRONICS: Exam will be written on paper by hand. No phones, computers, or calculators.

WRITING IMPLEMENT: Bring a bold-writing pen/pencil. Write prominently as your exam will be electronically scanned. You will not receive credit for any any answer that cannot be read.

REFERENCE SHEET: We will provide first two pages of the exam reference sheet.

SCRATCH WORK: You are NOT permitted to bring your own scrap paper. There will be enough room left on the exam for you to do some scratch work.

See your Canvas calendar (https://canvas.ubc.ca/calendar) for details.

Reference rule recap

Reference rule: When a data design **refers** to other non-primitive data, delegate the operation to a helper function.

- Must be applied **every** time a data design manipulates other data that is not primitive
- That includes List, Compound, Enumerated, and Optional types
- Anytime a variable is from a non-primitive type, we should invoke its template function

---

Helper functions

- A *helper function* is a normal function, but instead of solving the main problem, it solves a small part of the problem, helping the main function to solve the problem
- The main function is the function that actually solves the problem and uses the helper function to achieve this
- A good design has several small helper functions that do only a small task
- Every time the reference rule appears, it indicates that a helper function may be needed

⚠ Exception

No matter how easy the task is, must use a helper... **except** if the helper function would just return the **argument itself or a single field**. In that rare case, can access the argument or field directly, without calling a helper.

---

iClicker question

Consider the following data definition (just missing the template):

```
from typing import NamedTuple
CD = NamedTuple('CD',
                [('title', str),
                 ('artist', str),
                 ('price', MoneyAmount),
                 ('release_date', Date),
                 ('tracks', int) # in range[1,...)])
# interp. a CD for sale with its title, artist,
# price, release date, and number of tracks

CD1 = CD('Master Of Puppets', 'Metallica',
         MoneyAmount(16,75), Date(1986,3,3), 8)
```

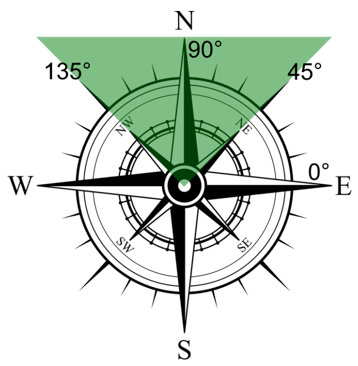
Without knowing anything more about the data types, for which fields will we need to apply the reference rule when writing the template?

Select ALL that apply. [Set question type to "Multiple Answer".]

A. `title`
B. `artist`
C. `price`
D. `release_date`
E. `tracks`

▶ ℹ Hint (for your review after class)

Review: Example 1 from last class



Problem: Given the compound data `Velocity`, write a function to compute the average speed of all velocities with a *northerly* heading in a list. We will consider directions in the range 45-135 degrees as northerly.

Last time we had completed designing the main function and just needed to finish our `is_northerly` helper function so we could test and debug it.

Data definitions for `Velocity` and `List[Velocity]`

In [ ]:
```python
from typing import NamedTuple
Velocity = NamedTuple('Velocity', [('speed', float),
                                   ('dir', int)]) # in range[0,359]

# interp. a velocity with its speed in m/s and direction
# as an angle in degrees with east=0 increasing counterclockwise

V1 = Velocity(9, 22)
V2 = Velocity(3.4, 180)

# template based on Compound (2 fields)
@typecheck
def fn_for_velocity(v: Velocity) -> ...:
    return ...(v.speed, v.dir)


from typing import List
# List[Velocity]
# interp. a list of velocities

LOV0 = []
LOV1 = [Velocity(3.1, 41)]
LOV2 = [V1, V2]
LOV3 = [Velocity(5.9, 265), Velocity(3.5, 89), Velocity(7.9, 323)]

@typecheck
# template based on arbitrary-sized and reference rule
def fn_for_lov(lov: List[Velocity]) -> ...:
    # description of the accumulator
    acc = ... # type: ...
    for v in lov:
        acc = ...(fn_for_velocity(v), acc)
    return ...(acc)
```

Example 1 Helper functions

Your completed helper function should look like this. Notice the "edge cases" in the examples.

In [ ]:
```python
@typecheck
def is_northerly(v: Velocity) -> bool:
    """
    Returns True if v has a direction in the range [45,135],
    otherwise False.
    """
    # return False # stub
    # template from Velocity
    return v.dir >= 45 and v.dir <= 135

start_testing()

expect(is_northerly(V1), False)
expect(is_northerly(V2), False)
expect(is_northerly(Velocity(1.2, 44)), False)
expect(is_northerly(Velocity(3.4, 45)), True)
expect(is_northerly(Velocity(5.6, 135)), True)
expect(is_northerly(Velocity(7.8, 136)), False)

summary()
```

Example 1 Main function

In [ ]:
```
@typecheck
def average_speed_of_northerly(lov: List[Velocity]) -> float:
    """
    Returns the average speed of all velocities in lov
    with a northerly direction.

    Northerly means dir is in the range [45,135].

    Returns 0.0 if the list doesn't have any northerly-directed
    velocities.
    """
    # return -1 # stub

    # template from List[Velocity]
    # sum_northerly is the sum of speeds of northerly velocities
    #   seen in the list so far
    sum_northerly = 0.0 # type: float
    # count_northerly is the count of northerly velocities
    #   seen in the list so far
    count_northerly = 0 # type: int

    for v in lov:
        # acc = ...(fn_for_velocity(v), acc)
        if is_northerly(v):
            sum_northerly = sum_northerly + v.speed
            count_northerly = count_northerly + 1

    if count_northerly == 0:
        return 0.0
    else:
        return sum_northerly / count_northerly


start_testing()

expect(average_speed_of_northerly(LOV0), 0.0)
expect(average_speed_of_northerly(LOV1), 0.0)
expect(average_speed_of_northerly(LOV3), 3.5)
expect(average_speed_of_northerly( [ Velocity(1, 44),
                                     Velocity(2, 45),
                                     Velocity(3, 135),
                                     Velocity(4, 136)
                                   ]), 2.5)

summary()
```

Multiple Helper Functions?

Last class, we were going to approach the same problem but with a bottom-up approach. Let's make a plan in English for how to proceed with that, but I'll leave the solution of that plan as an exercise! You can find a full solution in last class's notes, however.

Ian mentioned the sort of functions we might want would be:

- A function that finds the average speed of a list of velocities (already written in a previous class)
- A function that takes a list of velocities and returns a list of only the northerly velocities from that list
- A function that determines whether a velocity is northerly (already written last time)

So.. how should we put these together (in English for now, not in code)?

▶ ℹ️ Sample solution (For later. Don't peek if you want to learn 🙂)

Once you have a plan in English, modify `average_speed_of_northerly_v2` below to use that approach.

▶ ℹ️ Sample solution that uses three helper functions (For later. Don't peek if you want to learn 🙂)

In [ ]:
```
@typecheck
def average_speed_of_northerly_v2(lov: List[Velocity]) -> float:
    """
    Returns the average speed of all velocities in lov
    with a northerly direction.

    Northerly means dir is in the range [45,135].

    Returns 0.0 if the list doesn't have any northerly-directed
    velocities.
    """
    # return -1 # stub

    # template from List[Velocity]
    # sum_northerly is the sum of speeds of northerly velocities
    #   seen in the list so far
    sum_northerly = 0.0 # type: float
    # count_northerly is the count of northerly velocities
    #   seen in the list so far
    count_northerly = 0 # type: int

    for v in lov:
        # acc = ...(fn_for_velocity(v), acc)
        if is_northerly(v):
            sum_northerly = sum_northerly + v.speed
            count_northerly = count_northerly + 1

    if count_northerly == 0:
        return 0.0
    else:
        return sum_northerly / count_northerly


start_testing()

expect(average_speed_of_northerly_v2(LOV0), 0.0)
expect(average_speed_of_northerly_v2(LOV1), 0.0)
expect(average_speed_of_northerly_v2(LOV3), 3.5)
expect(average_speed_of_northerly_v2( [ Velocity(1, 44),
                                        Velocity(2, 45),
                                        Velocity(3, 135),
                                        Velocity(4, 136)
                                      ]), 2.5)

summary()
```

Reference rule applied to other data types

Lists can refer to other types defined in a data definition, but so can several other types of data. Specifically, Optionals and Compounds can refer to other data definitions. In those cases, you follow the same reference rule as with lists.

Every time you are manipulating non-primitive data, the reference rule applies. The template, if written correctly, will help you know when a function is needed.

In the case of a function for a `List[Velocity]` we applied the reference rule to refer to data of the non-primitive `Velocity` type. We didn't need to apply the reference rule to the template function for `Velocity` because it just refers to primitive data types. (We treat the interval `dir` as primitive).

$$\begin{tikzcd}
\text{List} \arrow[r] & \text{Velocity} \arrow[r] \arrow[dr] & \text{float (primitive)} \\
& & \text{int (primitive)}
\end{tikzcd}$$

Exercise 1a: A modified `Velocity`

Let's try working with a modified version of `Velocity` that contains a `Direction` as a cardinal compass direction (enumeration).

Now the compound refers to a non-primitive field:

$$\begin{tikzcd}
\text{List} \arrow[r] & \text{Velocity} \arrow[r] \arrow[dr] & \text{float (primitive)} & \\
& & \text{Direction} \arrow[r] & \text{distinct value (primitive)}
\end{tikzcd}$$

Let's see how we apply the reference rule for this modified data type.

The data definition for `Direction` is provided:

▶ Jump to...

In [ ]:
```python
from enum import Enum

Direction = Enum("Direction",["N", "E", "S", "W"])

# interpr. a direction (N - North, E - East, S - South, W - West)

# Examples are redundant for Enumeration

# template based on Enumeration (4 cases)
@typecheck
def fn_for_direction(d: Direction) -> ...:
    if d == Direction.N:
        return ...
    elif d == Direction.E:
        return ...
    elif d == Direction.S:
        return ...
    elif d == Direction.W:
        return ...
```

Fixing the data definition for `Velocity`

▶ Jump to...

In [ ]:
```python
# TODO: Fix the data definition for `Velocity` to use `Direction`
```

```python
from typing import NamedTuple
Velocity = NamedTuple('Velocity',
                      [('speed', float),
                       ('dir', int)]) # in range[0,359]

# interp. a velocity with its speed in m/s and direction
# as an angle in degrees with east=0 increasing counterclockwise

V1 = Velocity(9, 22)
V2 = Velocity(3.4, 180)

# template based on Compound (2 fields)
@typecheck
def fn_for_velocity(v: Velocity) -> ...:
    return ...(v.speed, v.dir)
```

▶ ℹ Sample solution (For later. Don't peek if you want to learn 🙂)

▶ Jump to...

Fixing the data definition for `List[Velocity]`

In [ ]:

```python
# TODO: Fix the data definition for `List[Velocity]` to use `Direction`

from typing import List
# List[Velocity]
# interp. a list of velocities

LOV0 = []
LOV1 = [Velocity(3.1, 41)]
LOV2 = [V1, V2]
LOV3 = [Velocity(5.9, 265), Velocity(3.5, 89), Velocity(7.9, 323)]

@typecheck
# Template based on arbitrary-sized and reference rule
def fn_for_lov(lov: List[Velocity]) -> ...:
    # description of the accumulator
    acc = ... # type: ...
    for v in lov:
        acc = ...(fn_for_velocity(v), acc)
    return ...(acc)
```

---

iClicker question: Where to apply the reference rule?

Target one of the places we'll need to change in order to apply the reference rule in the code above. [Set question type to "Target".]

▶ ℹ Solution (For later. Don't peek if you want to learn 🙂)

Exercise 1b: A similar problem

Problem: Given the modified compound data `Velocity`, write a function to compute the average speed of all velocities in a list with a *user-specified* `Direction` (i.e., a function argument).

Space reserved for additional helper functions

▶ Jump to...

In [ ]:
```
# TODO: Add any added/changed helper functions here after designing the main functi
```

▶ ℹ Sample helper functions (For later. Don't peek if you want to learn 🙂)

In [ ]:
```
# TODO: Add any added/changed helper functions here after designing the main functi
```

▶ ℹ Sample helper function (For later. Don't peek if you want to learn 🙂)

Modify our main function

▶ Jump to...

Problem: Given the modified compound data `Velocity`, write a function to compute the average speed of all velocities in a list with a *specified* `Direction`.

Our original `average_speed_of_northerly` function from Module 05b Day 1 is provided below. Let's modify it as needed to solve the new problem, using our modified `Velocity`. Add/edit any helper functions in the cell above.

In [ ]:
```
# TODO: Modify `average_speed_of_northerly` function as needed

@typecheck
def average_speed_of_northerly(lov: List[Velocity]) -> float:
    """
    Returns average speed of all northerly velocities
    in list lov.
    We define northerly as being between 45 and 135 degrees (inclusive).
    If no northerly velocities in list, then returns 0.
    """
    # return -1 # stub
    # template from List[Velocity]

    # sum of speeds of northerly velocities so far
    sum_speeds = 0 # type: float

    # count of speeds of northerly velocities so far
    count_speeds = 0 # type: int

    for v in lov:
        # acc = ...(fn_for_velocity(v), acc)
        if is_northerly(v):
            sum_speeds = sum_speeds + v.speed
            count_speeds = count_speeds + 1

    if count_speeds == 0:
        return 0

    return sum_speeds / count_speeds

start_testing()

expect(average_speed_of_northerly(LOV0), 0)
expect(average_speed_of_northerly(LOV1), 0)
expect(average_speed_of_northerly(LOV2), 0)
expect(average_speed_of_northerly(LOV3), 3.5)
# other examples:
# multiple northerly
# edge cases

summary()
```

▶ ℹ Sample solution (For later. Don't peek if you want to learn 🙂)

Summary

✅ Helper after helper

Recall how we nested non-primitive data types in our `List`:

$$\begin{tikzcd} \text{List} \arrow[r] & \text{Velocity} \arrow[r] \arrow[dr] & \text{float (primitive)} & \\ & & \text{Direction} \arrow[r] & \text{distinct value (primitive)} \end{tikzcd}$$

The reference rule required us to call a helper function in each respective template function:

`fn_for_lov` calls `fn_for_velocity` calls `fn_for_direction`

⚠ Comparing two enums - no template required

Notice that `is_same_dir` compares two enumerations (in this case, `Direction`). As we saw, in this particular case, the enumeration template function turns out not to be very useful.

In the special case of designing a function that compares two enumerations, you are permitted to not follow the template if it becomes clear that another approach

is simpler and clearer.

Modularity

- Notice that the `average_speed_of_northerly` function body didn't change (just examples did)
- Direction only stored in `Velocity` so only helper functions for `Velocity` affected
- In this case, just `is_northerly(v)`
- Pieces of the program are working independently
- Each data type is handled by specific functions
- Each function is responsible for one small task
- Program is broken down into separate components (data types and functions) that can be developed, tested, and maintained separately

---

iClicker question: Modular programming

What are some advantages of **modular programming**? Select ALL that apply.

A. It facilitates collaborative development by allowing developers to work on different modules at the same time
B. It promotes code reuse and modifiability
C. It makes it easier to test and debug code
D. It enables developers to write code faster without worrying about code structure
E. It allows developers to break down a software system into smaller, more manageable components

▶ Hint (for your review after class)

⚠ Exercise 2, if time allows

We will get to the following in class if time allows. Otherwise, you should complete them after class and compare your solutions to the sample solutions provided.

Exercise 2 demonstrates how the reference rule applies for an `Optional[non-primitive]`

- 2a demonstrates HtDD and
- 2b demonstrates HtDF.

Exercise 2a: Listed or unlisted store price

Problem: Given a data definition to represent an amount of money, `MoneyAmount`, in dollars and cents, design another data definition to represent an item's price in a store, which may be listed or unlisted (for example, if the price is negotiable).

Data definition for `MoneyAmount`

▶ Jump to...

In [ ]:
```
from typing import NamedTuple

MoneyAmount = NamedTuple('MoneyAmount', [('dollars', int), # in range [0, ...)
                                         ('cents', int)])  # in range [0, 100)
# interp. an amount of money in dollars and cents.
MA0 = MoneyAmount(0, 0)
MA2_50 = MoneyAmount(2, 50)
MA99_99 = MoneyAmount(99, 99)

# template based on compound (2 fields)
@typecheck
def fn_for_money_amount(ma: MoneyAmount) -> ...:
    return ...(ma.dollars,
               ma.cents)
```

Data definition for item's price in a store

▶ Jump to...

In [ ]:
```
# TODO: Design Price data definition
```

▶ ℹ Sample solution (For later. Don't peek if you want to learn 😳)

Exercise 2b: Can I afford it?

Problem: Design a function to determine if I might be able to afford an item based on my amount of money and its price.

Helper functions (to be filled in as needed)

▶ Jump to...

In [ ]:
```
# TODO: Add any helper functions here after designing the main function below
```

▶ ℹ️ Sample solution (For later. Don't peek if you want to learn 🙄)

▶ Jump to...

Main function (start here for top-down approach)

In [ ]:
```
# TODO: Design main function here first (top-down approach)
```

▶ ℹ️ Sample solution (For later. Don't peek if you want to learn 🙄)