Programming Paradigms*: Intro

Instructor: Doug Howe, douglashowe@cunet.carleton.ca

* paradigms = {functional}

What's in a programming Language?

Data

- constants (17, "foo"...)
- arrays
- objects (struct)
- variable = memory location
- relational databases
- lists

Control

- functions
- if-then-else
- loops
- semaphore
- exceptions
- type casts?
- eval of expressions
- assignments
- allocation

Scan of current languages

Data

Control

The "procedural", or "imperative" programming paradigm

Central concept: **Modifying state**.

The *state* of a computation comprises:

- values of variables
- content of accessible memory
- current location(s) in program(s)

Essentially, a program *executes* a sequence of assignments that modify state.

Data: laid out in memory

Control: commands/statements that explicitly modify state, e.g. assignment, memory allocation.

A common imperative programming pattern

Pattern: traverse a data structure, making local modifications.

```
# Binary trees with integer data
class Node:
  def __init__(self, data):
    self.left = None
    self.right = None
    self.data = data

# Use pattern to increment all data by 1
def incr(t):
  if t is None:
    return
  else:
    t.data += 1
    incr(t.left)
    incr(t.right)
    return
```

Key concept of functional programming

Central concept: **values of expressions.**

- variable values never change
- data structures are *immutable*, i.e. can't be modified

Data: Abstract view

- constructors and accessors/destructors only
- don't care how data is represented in memory

Control: Function calls

- computation power comes from *recursion*

A functional analog of the imperative pattern example

Abstract view of trees

- Constructors: *empty, node(n, t1, t2)*
- Accessors: *data(t), left(t), right(t)*

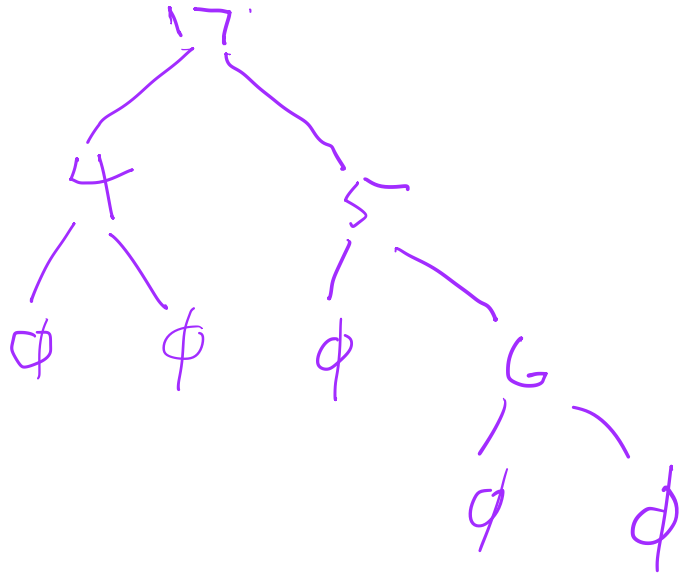
  Discriminator: *is_empty(t)*

What does "abstract" mean here?

- We don't know how the functions *node, data, left, right,* etc are implemented
- We *do* know the usual properties, e.g. *left(node(n,t1,t2)) = t1*

Example of "abstract" tree

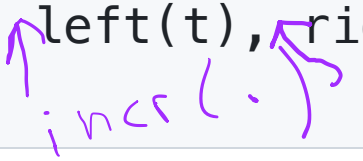
node(17, node(4,empty,empty), node(5,empty,node(6,empty,empty)))



Increment data field: functional version

We'll use a "pseudo-Python" where everything is an expression

```
def incr(t):
  # "return" value of incr is value of this if-then-else *expression*:
  if is_empty(t):
    # if condition true, if-then-else has value t
    t
  else:
    # otherwise it has as value this:
    node(data(t)+1, left(t), right(t))
```



Summary: imperative vs functional

Imperative: "do", "modify", "set", "update", "deallocate", "initialize"

Functional: "evaluate", "name", "construct", "access"

Why study/use functional programming style/languages?

- avoid big classes of common errors: pointer, memory de/allocation, aliasing, concurrency
- modern languages have big functional components, e.g. closures in js
- much closer to mathematics: easier to reason about since it abstracts away from implementation details

Course structure

Content:

- At least 3/4 of the course will be on Haskell
- At most 1/4 of the course will be on Go

Quizzes and final exam:

- 90% of your grade
- Quizzes + exams will be **all** autograded coding

Assignments:

- 10% of grade
- quiz/exam questions will be **directly based** on assignments

Course communication and help

Ed: announcements, Q&A, defined hours (TBA) for TA participation

Gradescope: assignment distribution, submission and grading

Brightspace: nothing

TA office hours: for general tutoring, especially for struggling students