Programming Languages and Techniques (CIS120)

Lecture 5

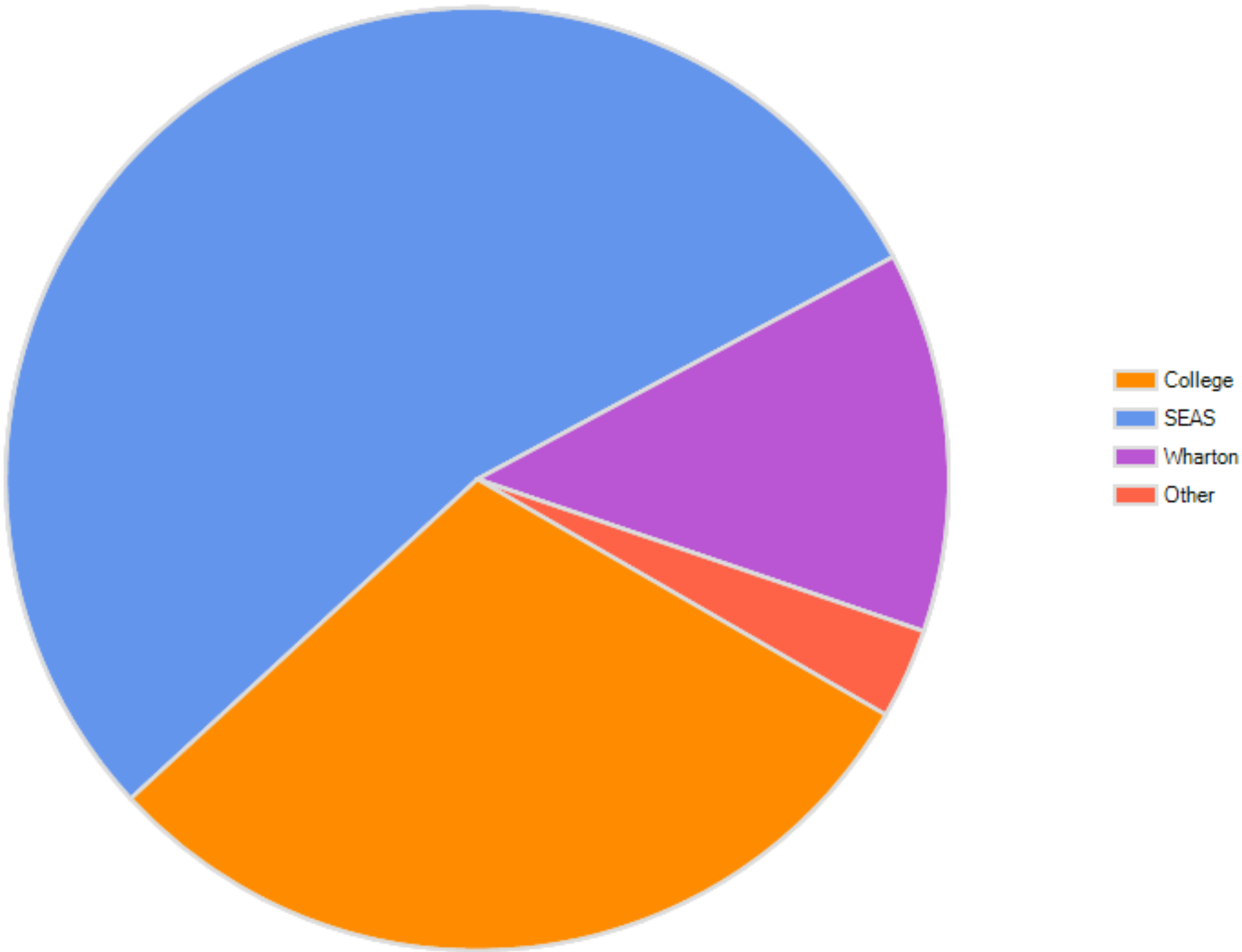
Jan 18, 2013

Tuples and Lists

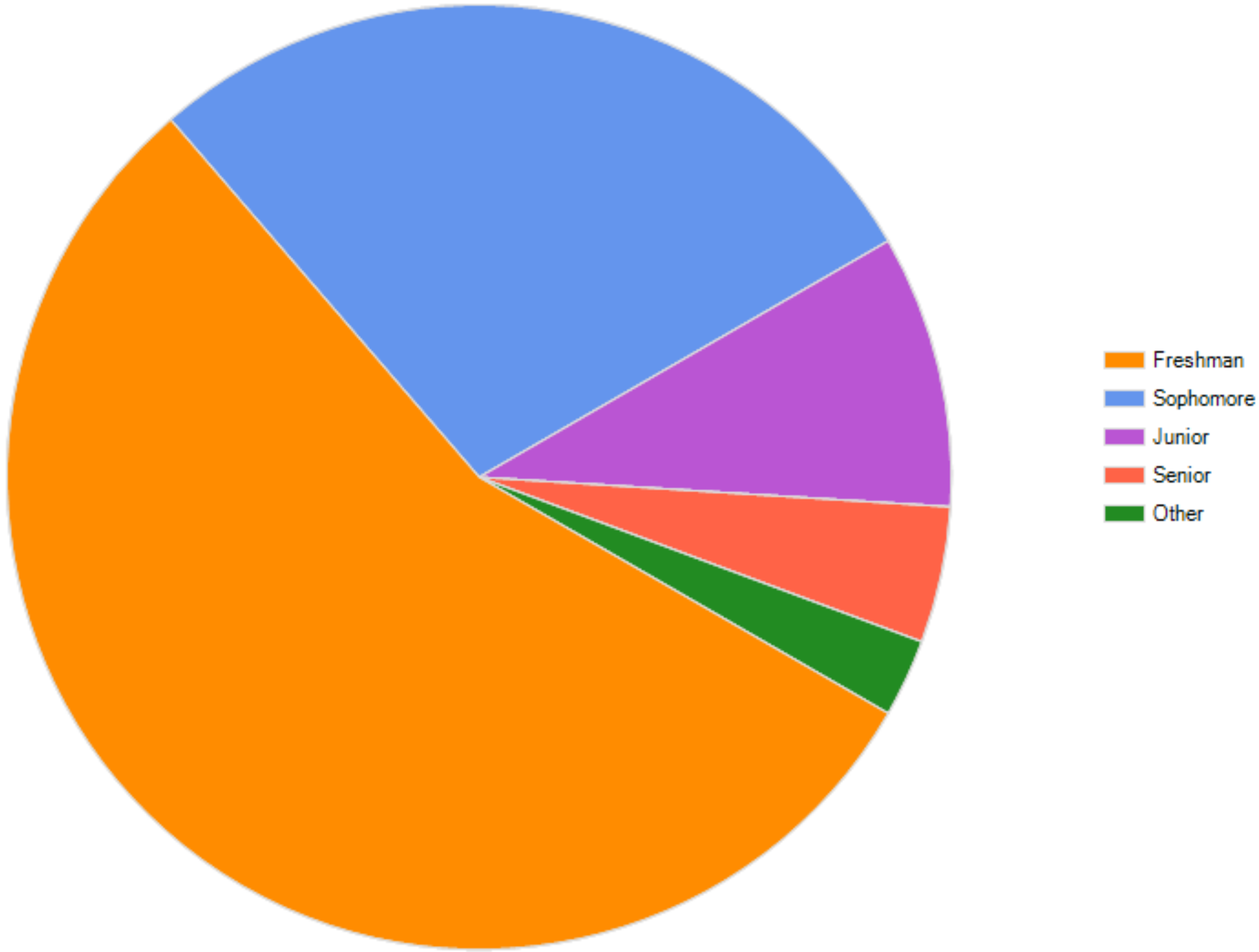
Announcements

- No class Monday
- Homework 1 due Tuesday at midnight
  - Don’t wait! 24 people have already submitted
- See schedule for TA office hours Sunday, Monday and Tuesday
- Post to piazza for help
- Weirich Monday OH moved to Wednesday

School



Year



CIS 120 Demographics

- 149 responses / ~190 registered
- 2/3 Male, 1/3 Female
- 80% taken CIS 110, 88% have CIS 110 or AP
- Java/C# experience

None	10s	100s	1000s	more
3.4% (5)	**27.0% (40)**	**61.5% (91)**	**6.8% (10)**	**1.4%(2)**

- Only **1 person** with ML/Haskell experience
- Python or Ruby experience

71.1% (86)	**19.8% (24)**	**7.4% (9)**	**0.8% (1)**	**0.8% (1)**

Tuples and Patterns

Tuples

- A tuple is a way of grouping together two or more data values (of possibly different types).
- In OCaml, tuples are created by writing the values, separated by commas, in parentheses:

```
let my_pair = (3, true)
let my_triple = (“Hello”, 5, false)
let my_quaduple = (1,2,”three”,false)
```

- Tuple types are written using ‘*’
  - e.g. `my_triple` has type:

```
string * int * bool
```

Pattern Matching Tuples

- Tuples can also be inspected by pattern matching:

```
let first (x: string * int) : string =
  begin match x with
  | (left, right) -> left
  end

first (“b”, 10)
⇒
“b”
```

- Note how, as with lists, the pattern follows the syntax for the corresponding values

Mixing Tuples and Lists

- Tuples and lists can mix freely:

```
[(1,"a"); (2,"b"); (3,"c")]
               : (int * string) list
```

```
([1;2;3], ["a"; "b"; "c"])
               : (int list) * (string list)
```

Nested Patterns

- So far, we've seen simple patterns:

```
[]
x::tl
(a,b,c)
```

- Like expressions, patterns can *nest*:

`x::(y::tl)` *matches lists of length at least 2*

`(x::xs, y::ys)` *matches pairs of non-empty lists*

- A useful pattern is the wildcard pattern: _

`_::tl` *matches a non-empty list, but only names tail*

`(_,x)` *matches a pair, but only names the 2nd part*

Lists and Tuple Examples

see lists.ml

Example: zip

- zip takes two lists of the same length and returns a single list of pairs:

  `zip [1; 2; 3] [“a”; “b”; “c”]` ⇒

  `[(1,”a”); (2,”b”); (3,”c”)]`

```
let rec zip (l1:int list)
            (l2:string list) : (int * string) list =
begin match (l1, l2) with
 | ([], []) -> []
 | (x::xs, y::ys) -> (x,y)::(zip xs ys)
 | _ -> failwith "zip: unequal length lists"
end
```

Unused Branches

- The branches in a match expression are considered in order from top to bottom.
- If you have "redundant" matches, then some later branches might not be reachable.
  - OCaml will give you a warning

```
let bad_cases (l : int list) : int =
  begin match l with
  | [] -> 0
  | x::_ -> x
  | x::y::tl -> x + y      (* unreachable *)
  end
```

This case matches more lists than that one does.

Exhaustive Matches

- Case analysis is *exhaustive* if every value being matched against can fit some branch's pattern.
- Example of a *non-exhaustive* match:

```
let sum_two (l : int list) : int =
  begin match l with
  | x::y::_ -> x+y
  end
```

- OCaml will give you a warning and show an example of what isn't covered by your cases.
  - in this example, there is no case for [], or for a singleton list
- The wildcard pattern and failwith are useful tools for ensuring match coverage.

More List Examples

see lists.ml

Recursive function patterns

Recursive functions over lists follow a general pattern:

```
let rec number_of_songs (pl : string list) : int =
  begin match pl with
  | [] -> 0
  | ( song :: rest ) -> 1 + number_of_songs rest
  end
```

```
let rec contains (pl:string list) (s:string) : bool =
  begin match pl with
  | [] -> false
  | ( song :: rest ) -> s = song || contains rest s
  end
```

Structural Recursion Over Lists

Structural recursion builds an answer from smaller components:

```
let rec f (l : … list) … : … =
  begin match l with
  | [] -> …
  | ( hd :: rest ) -> … f rest …
  end
```

The branch for `[]` calculates the value (`f []`) directly.

The branch for `hd::rest` calculates (`f(hd::rest)`) given `hd` and (`f rest`).

Design Pattern for Recursion

1. Understand the problem
   What are the relevant concepts and how do they relate?
2. Formalize the interface
   How should the program interact with its environment?
3. Write test cases
   - If the main input to the program is an immutable list, make sure the tests cover both empty and non-empty cases
4. Implement the required behavior
   - If the main input to the program is an immutable list, look for a recursive solution…
     - Suppose someone has given us a partial solution that works for lists up to a certain size. Can we use it to build a better solution that works for lists that are one element larger?
     - Is there a direct solution for the empty list?