CS162 Week 2

Kyle Dewey

Overview

- Continuation of Scala
- Assignment 1 wrap-up
- Assignment 2a

Scala Continued

Traits

- Created with the `trait` reserved word
- Like a mixin in Ruby
- Think Java interfaces, but they can have methods defined on them
  - More powerful than that, but not relevant to this course

-See trait.scala

`object`

- Used in much the same way as `static` is in Java
- Defines both a class and a single instance of that class (and **only** a single instance)
- Automated implementation of the Singleton design pattern
- Keeps everything consistently an object

-See object.scala and object.java

`equals`, `==`, **and** `eq`

- As with Java, if you want to compare **value** equality, you must extend `equals`
  - Case classes automatically do this for you
- However, instead of saying `x.equals(y)`, **merely say** `x == y`
- If you want **reference** equality, say: `x eq y`

Case Classes

- Behave just like classes, but a number of things are automatically generated for you
  - Including `hashCode`, `equals`, and getters
- Typically used for pattern matching

-Regular classes can be used in pattern matching as well, but you usually need to put more effort into it (see the unapply method)
-See case_class.scala

Pattern Matching

- Used extensively in Scala
- Like a super-powerful `if`
- **Used with the `match` reserved word, followed by a series of `case`s**

-See pattern_matching.scala

`null`

- In general, `null` is an excellent wonderful/ terrible feature
- Often poorly documented whether or not `null` is possible
  - Checking for impossible cases
  - Not checking for possible cases

`Option`

- A solution: encode `null` as part of a type
- For some type, say `Object`, if `null` is possible say we have a `NullPossible<Object>`
- Scala has this, known as `Option`
- In general, if `null` is possible, use `Option`

-See option.scala

Tuples

- For when you want to return more than one thing
- Can be created by putting datums in parenthesis
- Can pattern match on them

-See tuples.scala, tuples_revisited.scala

Sequence Processing Functions

AKA: Why `while` is rare and `for` isn't `for`

Looping

- Scala has a `while` loop, but its use is highly discouraged (again, point loss)
  - It's not actually needed
- General functional programming style is recursion, but this is usually overkill

`foreach`

- Applies a given function to each element of a `Seq`

-See foreach.scala

map

- Like `foreach`, in that it applies a given function to each element of a sequence
- However, it also returns a new sequence that holds the return values of each of the function calls

-See MapExample.scala, MapExample.java

filter

- Takes a predicate, i.e. a function that returns true or false
- Applies the predicate to each item in a list
- A new list is returned that contains all the items for which the predicate was true

-See FilterExample.java, FilterExample.scala

`foldLeft`

- Extremely flexible, but sometimes unwieldy
- Takes a base element
- Takes a function that takes a current result and a current list element
- The function will manipulate result with respect to the current element

-See fold.scala, FoldLeftExample.java, FoldLeftExample.scala

`flatMap`

- Like `map`, but made especially for functions that return `Seq`s
- Will internally "flatten" all of the inner `Seq`s into a single `Seq`
- More on this later in the course

-see flatMap.scala

`for` Comprehensions

- Much like Python's list comprehensions
- Internally translated into a series of `foreach`, `flatMap`, `map`, **and** `filter` **operations**

-See for.scala

Assignment 1 Questions

- Level of challenge?
- Any problems worth going over?
- Pre/post order traversals?
- Run length encoding?

Assignment 2a: Logic Programming

Assignment Overview

- Goal: gain some familiarity with logic programming
  - Part A: Write code to traverse an ontology
  - Part B: Write a Sudoku solver

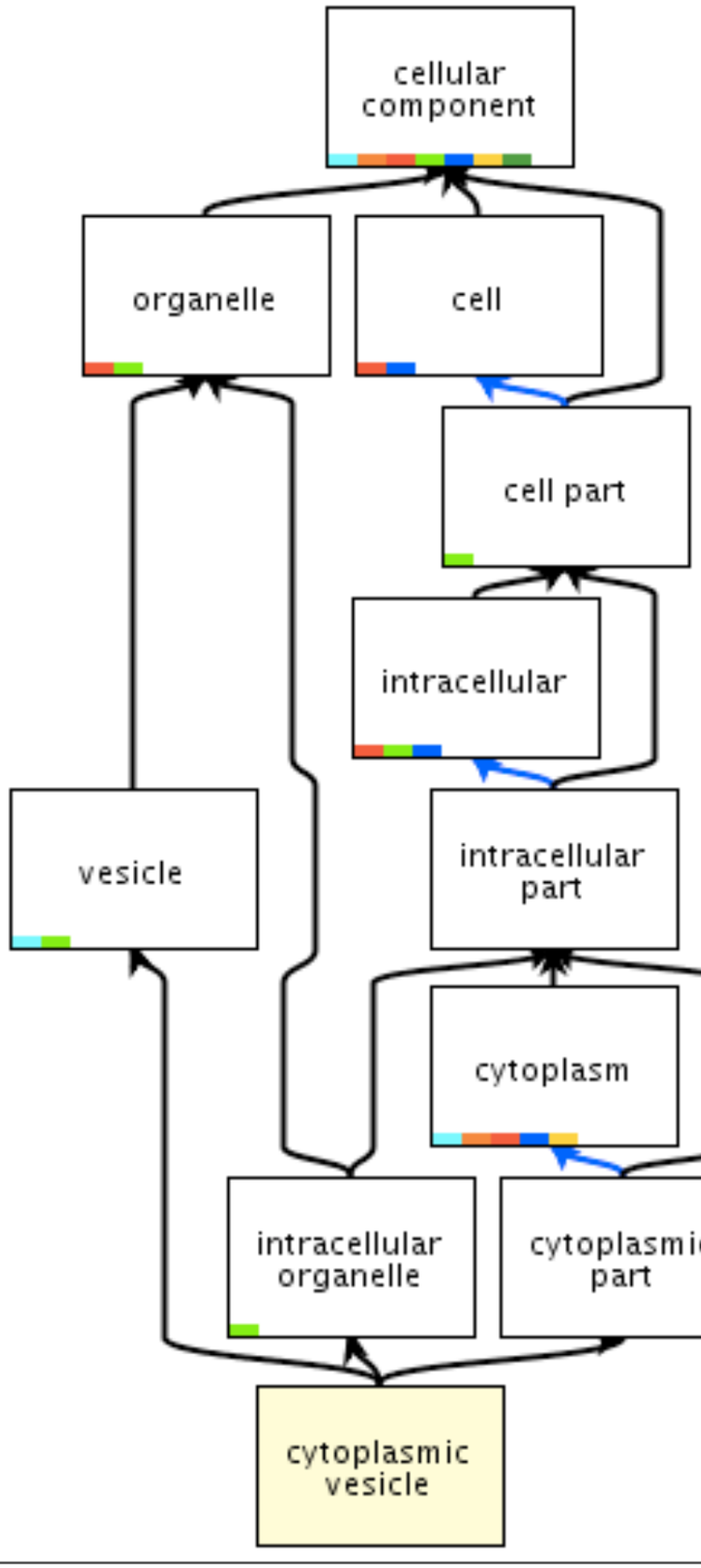
Ontologies

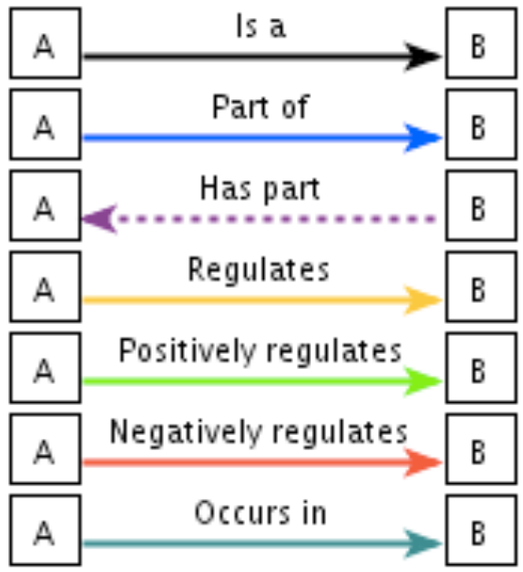
- Formal knowledge base
- Concepts and relationships between concepts
- Often representable as a giant, heavily annotated directed acyclic graph

The Gene Ontology

- Of great importance within Biology and Bioinformatics
- Database of biological processes, cellular components, and molecular function
- Allows scientists to quickly determine relevant information

Example





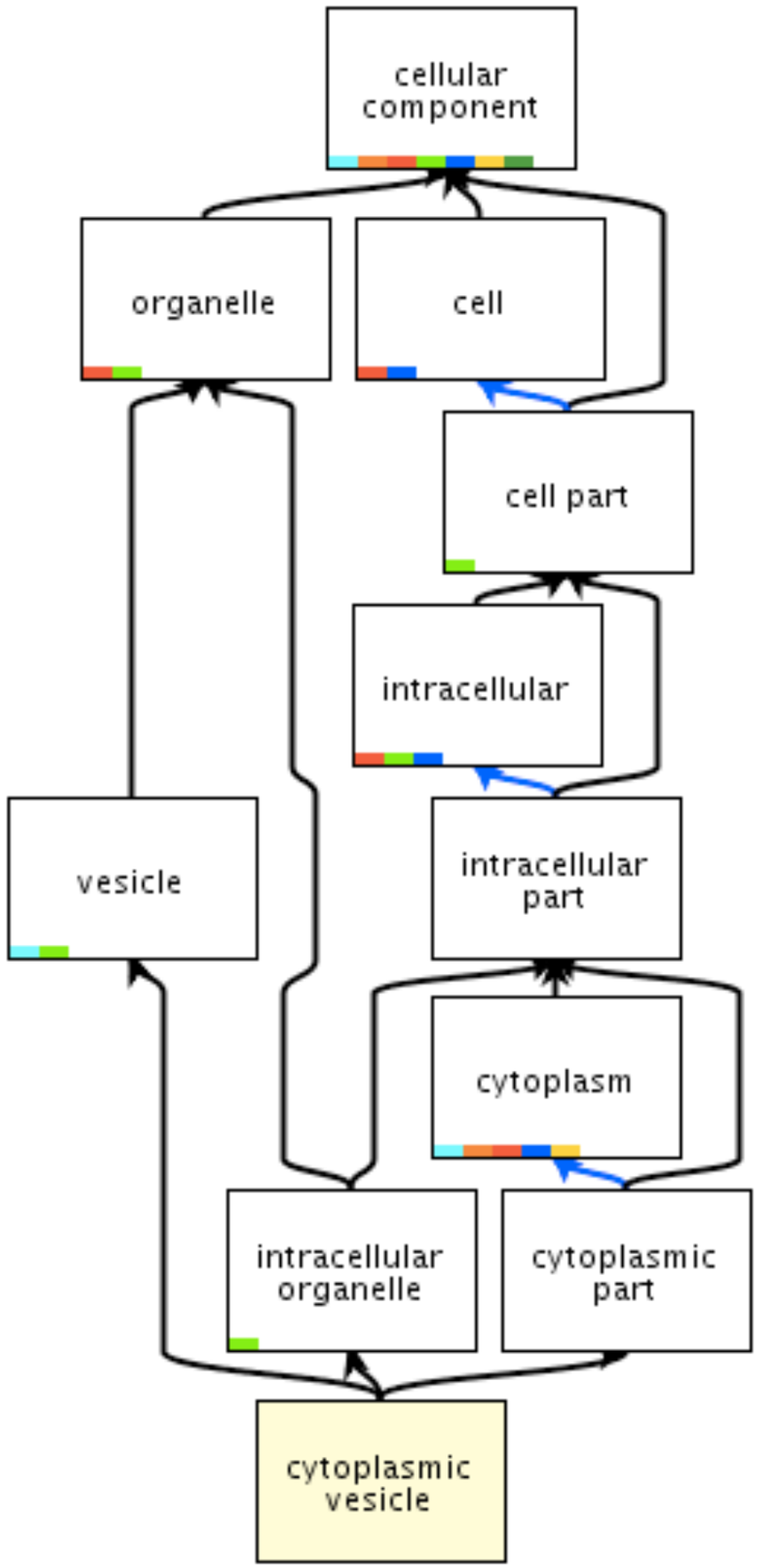
Friday, January 17, 14

-Note: intracellular part is not actually as cell part according to the GO, but it is rendered that way due to how the GO looks at part_of relationships (we ignore this here)

What to Write

- "A procedure named `toisaroot`, which makes a list showing the path from some given concept `C` to the root, which is always `'cellular component'`."
- For a given concept, there may be no path to the root, or multiple paths

isaroot Example



Friday, January 17, 14

-For example, a cytoplasmic vesicle can reach ‘cellular component’ in multiple ways in this diagram. All are possible answers. Each path starts with ‘cytoplasmic vesicle’ and ends at ‘cellular component’.

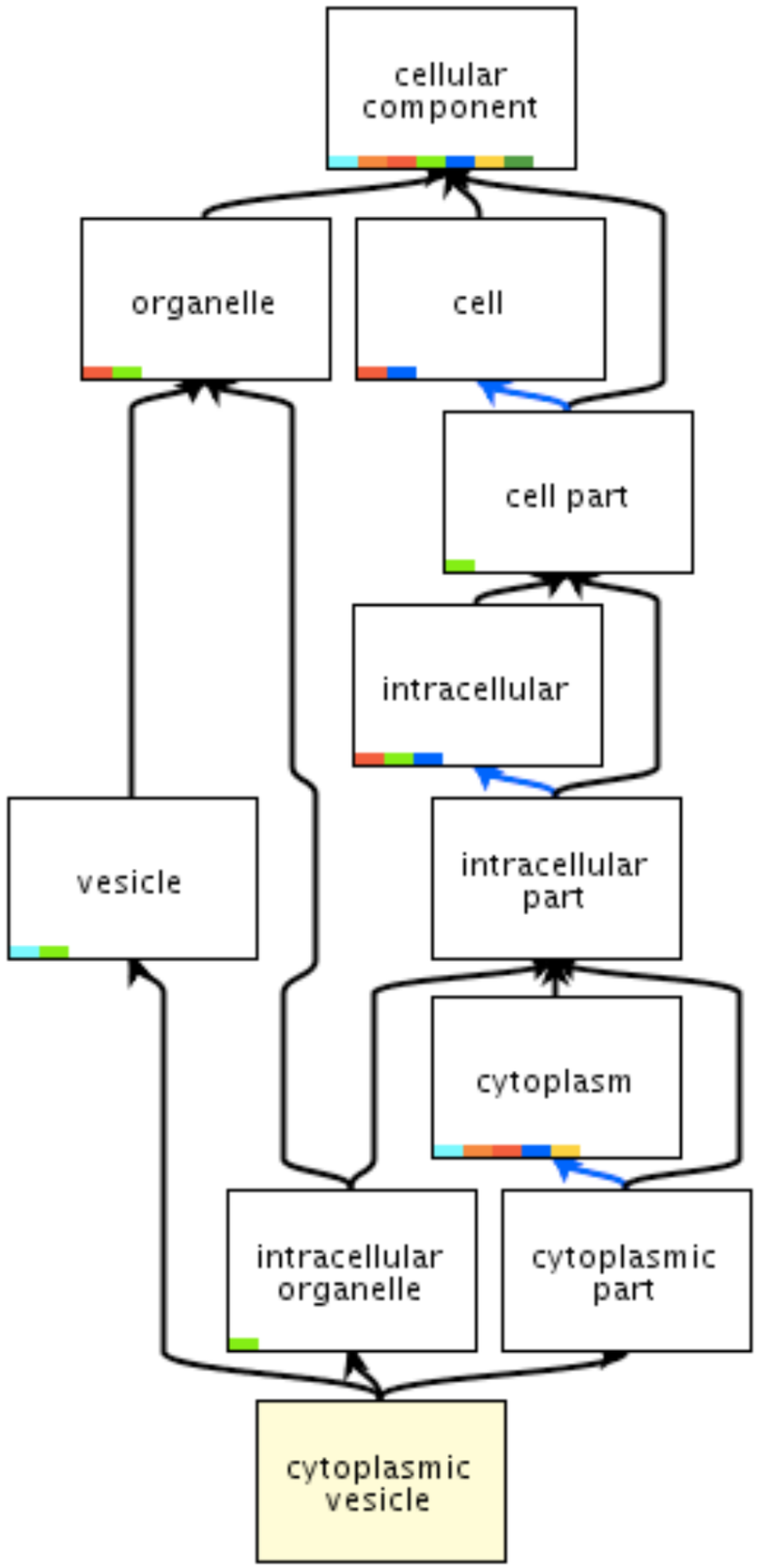
Another GO Search

- The GO has both `is_a` and `part_of` relationships
- This distinction may not always be important for gathering information of interest
- We may be interested in transitive relationships

What to Write

- "A procedure named `subconcept` to determine if a given concept `C1` is a subconcept of some concept `C2`. This is true if at least one of the following holds:"
  - `C1` **is a** `part_of` `C2`
  - `C1` `is_a` `C2`
  - The above two conditions hold transitively
- Another view: `C1` is a child node of `C2`

subconcept Example



Friday, January 17, 14

-For example, for the query subconcept('cytoplasm', X), X could be 'intracellular part', 'intracellular', 'cell part', cell, or 'cellular component', since these are all parents via either is_a or part_of relationships

Part B: Sudoku Solvers

- Sudoku: 9x9 board consisting of 9 3x3 subboards
- Each row, column, and subboard must contain the digits 1-9 (no repeats possible, and all must be accounted for)
- Some numbers are already given
- Goal: fill in the rest

Example - Given

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Example - Solution

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Goal

- Write a predicate `solve(Board)` that solves a 9x9 board
  - List of nine lists, each of which is a list of nine elements, where each element is either a number or an unknown (unbound variable)

Optimization

- Naive solution: try all possible combinations, and check constraints at the end
- Optimization: try one number, and ensure that constraints have not been violated
- For full credit, you **must** implement this optimization

Performance

- The solver will likely not perform well on hard boards, even with the optimization
- This will be addressed later on when we get into **constraint** logic programming
  - Will likely offer a cleaner solution, too