CSCI 334:
Principles of Programming Languages

Lecture 23: Exam Review

Instructor: Dan Barowy

Williams

Announcements

Thanks!

Announcements

Exam Study Session:
Monday, May 14 2-4pm
TBL 202

Announcements

HW9 solutions after late days

Announcements

If you are missing a grade, let me know!

Announcements

SCS Forms and Blue Sheets
at end of class
(10:50 for S1, 12:15 for S2)

Computability

i.e., what can and cannot
be done with a computer

def: a function *f* is **computable** if there
is a program *P* that computes *f*.

In other words, for **any** (valid) input *x*, the
computation *P(x)* **halts** with output *f(x)*.

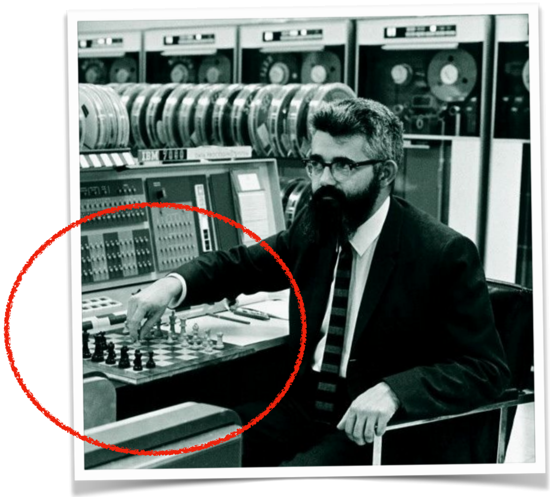
The Halting Problem

Decide whether program P halts on input x.

Given program P and input x,

$$\texttt{Halt(P,x)} = \begin{cases} \text{print } \texttt{"halts"} \text{ if } \texttt{P(x)} \text{ halts} \\ \text{print } \texttt{"does not halt"} \text{ otherwise} \end{cases}$$

Fun fact: it is provably impossible to write `Halt`



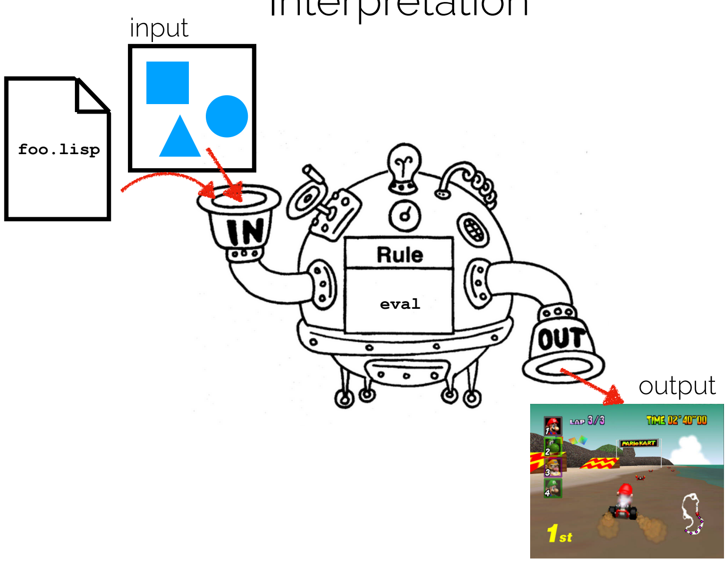
Lisp was invented for AI research

Higher-Order Functions

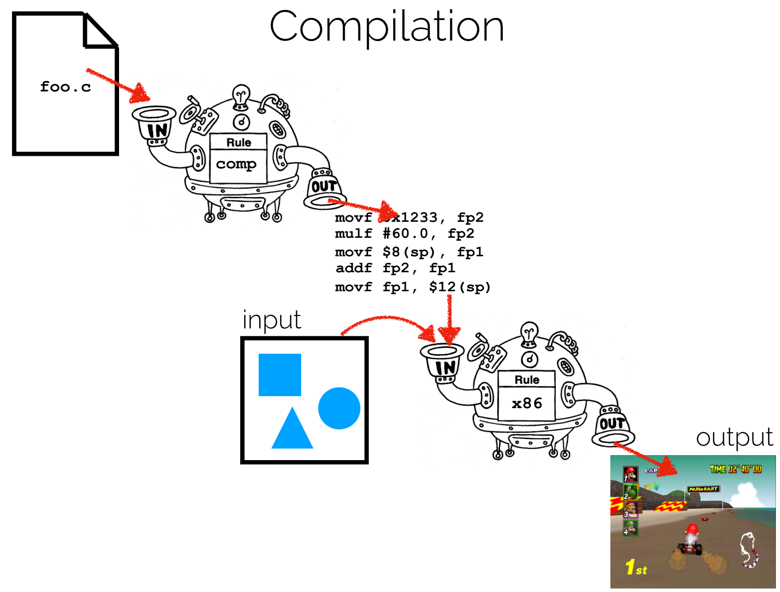
i.e., “functions that take functions”

```
(mapcar #'function list)
```

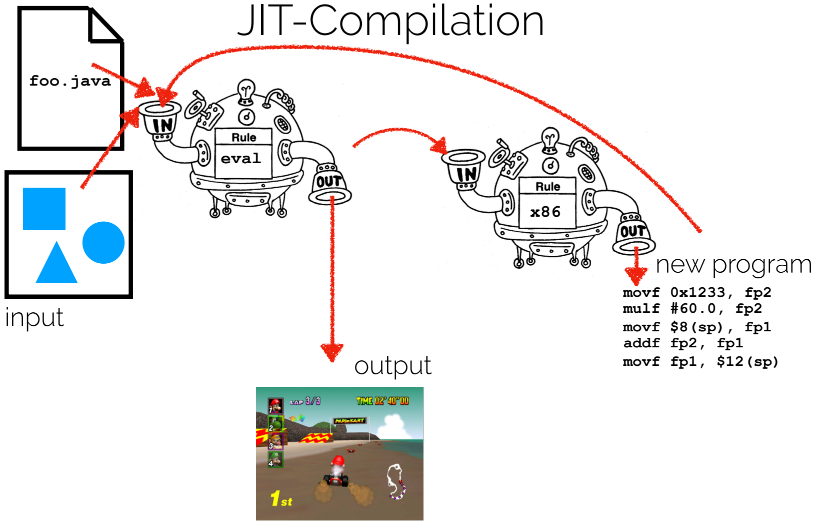
Interpretation



Compilation



JIT-Compilation



Lambda calculus

- Invented by Alonzo Church in order to solve the Entscheidungsproblem.
- Short answer to Hilbert's question: no.
- Proof: No algorithm can decide equivalence of two arbitrary **λ**-calculus expressions.

Lambda calculus is deceptively simple

- Church-Turing thesis: every computable function can be represented in the **λ**-calculus; i.e., it is "Turing complete".
- Grammar in BNF:

`M ::= x`		variable
`|`	`λx.M`	abstraction
`|`	`MM`	function application

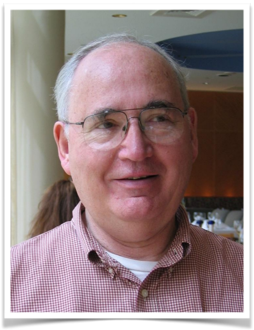
Order does not matter

$$\begin{tikzcd} & M \arrow[dl] \arrow[dr] & \\ M_1 \arrow[dr, dotted] & & M_2 \arrow[dl, dotted] \\ & N & \end{tikzcd}$$

If $M \to M_1$ and $M \to M_2$
then $M_1 \to^* N$ and $M_2 \to^* N$
for some N

"confluence"

ML



- Dana Scott
- Logic of Computable Functions (LCF)
  - Automated proofs!
  - Theorem proving is essentially a "search problem".
  - It is (essentially) NP-Complete
  - But works "in practice" with the right "tactics"

ML

- Robin Milner
- How to program tactics?
- A "meta-language" is needed
- ML is born

Static vs. dynamic environments

```
fun add_one x = x + 1
```

Static environment:

Facts about a program that are always true.

E.g., data types.

Other static facts:

- "always halts"
- fn is named "add_one"

Static vs. dynamic environments

```
fun add_one x = x + 1
```

Dynamic environment:

Facts about a program that are true for a given invocation of the program.

E.g., values.

Other dynamic facts:

- "halts for given value"

Types

We usually can determine types statically.

Some languages where we do:

Java, Standard ML, Go, Rust, …

Some languages where we don't:

Python, Ruby, Lisp, R, …

Nominal Types

Types are equivalent if they use the same *name* or if there is an explicit *subtype relationship* between names.

Matching names

```
int n = 3;
int m = 4;
n == m;
false
```

Subtype relationship

```
class Animal …
class Cat extends Animal …
Animal a = new Animal();
Cat c = new Cat();
c.equals(a) == true
```

(maybe)

Structural Types

Types are equivalent if they have the same features. Base case in ML: same name; inductive case: same composition of names.

Matching names

```
val n = 3
val m = 4
n = m
false
```

Structural relationship

```
val a = (1,(2,"hi"))
val b = (1,(2,"hi"))
a = b
true
```

algebraic datatypes and pattern matching (the chocolate and peanut butter of PL)

```
datatype treat =
   SNICKERS
 | TWIX
 | TOOTSIE_ROLL
 | DENTAL_FLOSS

fun trick_or_treat SNICKERS     = "treat!"
  | trick_or_treat TWIX         = "treat!"
  | trick_or_treat TOOTSIE_ROLL = "treat!"
  | trick_or_treat DENTAL_FLOSS = "trick!"
```

type checking

```
fun f(x:int) : int = "hello " + x
```

```
stdIn:27.12-27.24 Error: operator and operand don't
agree [overload conflict]
  operator domain: [+ ty] * [+ ty]
  operand:         string * int
  in expression:
    "hello " + x
```

Hinley-Milner algorithm

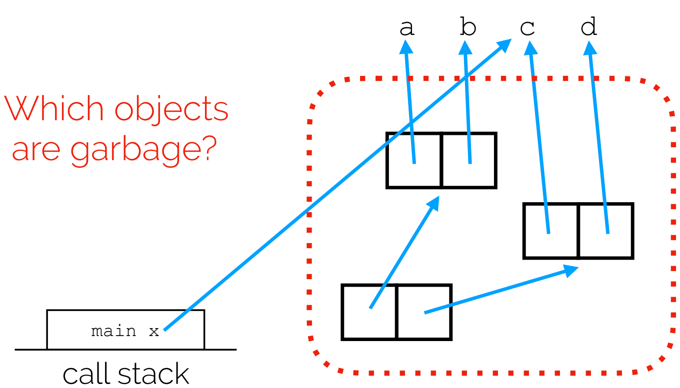
Has three main phases:

1. Assign type to each expression and subexpression
2. Generate type constraints based on rules of λ calculus:
   a. Abstraction constraints
   b. Application constraints
3. Solve type constraints using unification.

GC example from HW2

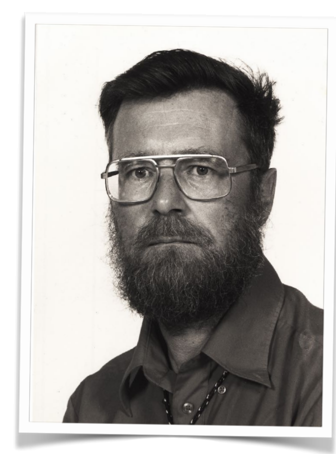
```
(car (cdr (cons (cons a b) (cons c b))))
```

Which objects are garbage?



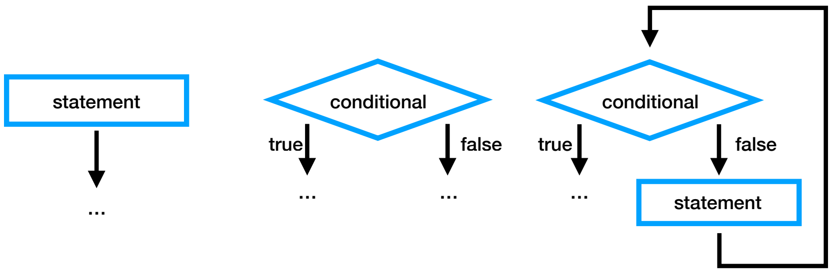
Structured Programming

- Coined by Edsger Dijkstra
- "GOTO Statement Considered Harmful"
- Argued that GOTO made programming much harder to understand.
- "the quality of programmers is a decreasing function of the density of GOTO statements in the programs they produce."



Structured Programming

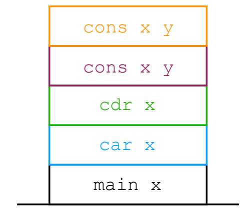
Only 3 building blocks for programs.



sequence conditional loop

Structured Program Theorem: Blocks are Turing-complete

Structured programs can be evaluated using a call stack



call stack

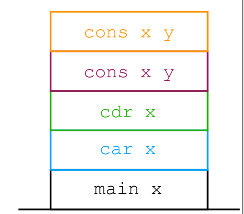
Call Stack

A *call stack* is a control structure that stores information about the active subroutines of a program.

Most programming language runtimes use a call stack to evaluate a program instead of evaluation-by-substitution (i.e., λ-calculus reductions).

Stacks are used to track…

1. which function is being executed *now*,
2. the parameters to that function,
3. the local variables used in that function,
4. temporary results needed along the way,
5. where to return when done,
6. where to put the result when done,
7. where to find non-local variables (optional)



call stack

What can a function return?

First Class Functions

- A language with *first-class functions* treats functions no differently than any other value:
- You can assign functions to variables:

```
val f = fn x => x + 1
```

- You can pass functions as arguments:

```
fun g h = h 3
g f
```

- You can return functions:

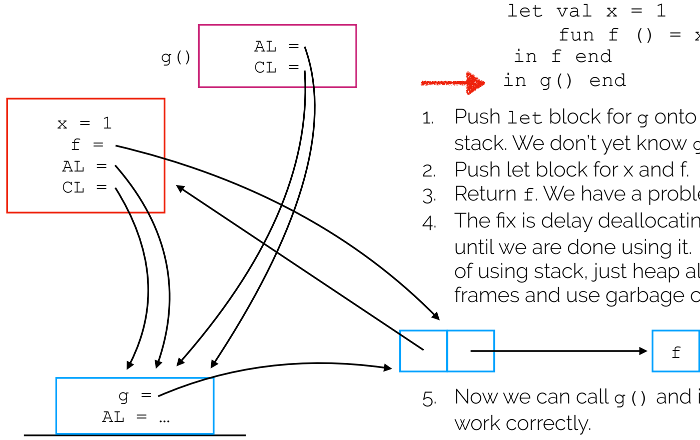
```
fun k x = fn () => x + 3
```

- First-class function support complicates *implementation* of lexical scope.

Upward funargs

```
let val g =
  let val x = 1
      fun f () = x + 1
   in f end
  in g() end
```

1. Push `let` block for `g` onto call stack. We don't yet know `g`'s value.
2. Push let block for x and f.
3. Return `f`. We have a problem!
4. The fix is delay deallocating record until we are done using it. Instead of using stack, just heap allocate frames and use garbage collector!



5. Now we can call `g()` and it will work correctly.

heap-allocated records

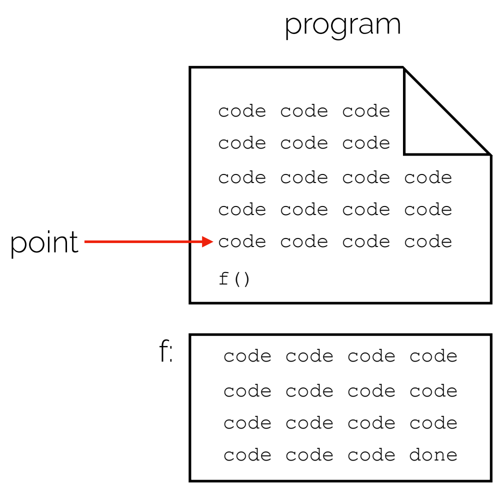
Exceptions are dynamically scoped

- Remember: variable bindings are statically (lexically) scoped.
- Exceptions are *dynamically scoped*.

```
fun prod (Leaf x) =
   if x = 0 then raise Zero else x
  | prod (Node(x,y)) = prod x * prod y
```

- Remember that I said `raise` is like `goto`?
- Where would this raise "go to"? We haven't even used `prod` yet!

Continuations



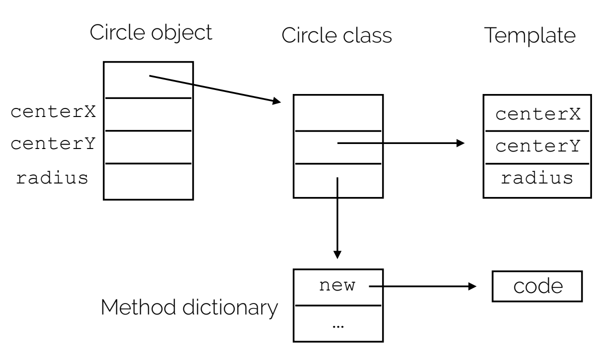
OK, really, what is OO?

Object-oriented programming is composed primarily of four key language features:

1. Abstraction
2. Dynamic dispatch
3. Subtyping
4. Inheritance

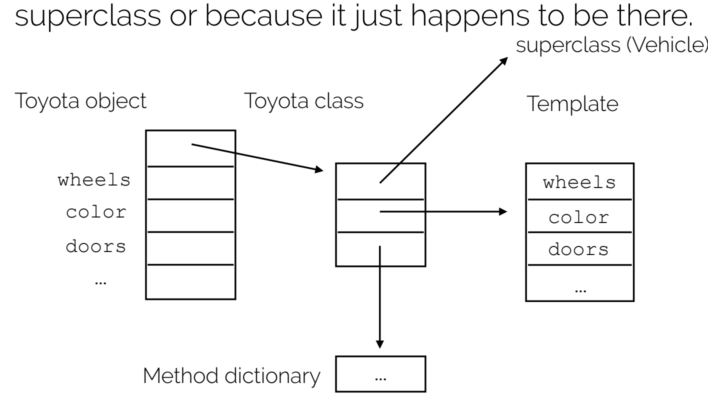
Dynamic Dispatch

- Dynamic dispatch is an algorithm for finding an object's method corresponding to a given selector name.



Polymorphism

- Dynamic dispatch allows for a kind of polymorphism.
- It does not matter whether a method exists because of a superclass or because it just happens to be there.



OO vs Functional Tradeoff

- OO offers a different kind of extensibility than functional (or function-oriented) languages.
- Suppose you're modeling a hospital.

Operation	Doctor	Nurse	Orderly
`Print`	`Print Doctor`	`Print Nurse`	`Print Orderly`
`Pay`	`Pay Doctor`	`Pay Nurse`	`Pay Orderly`

- Functional programming makes it easy to add operations.
- OO programming makes it easy to add data.

Subtyping vs. Inheritance

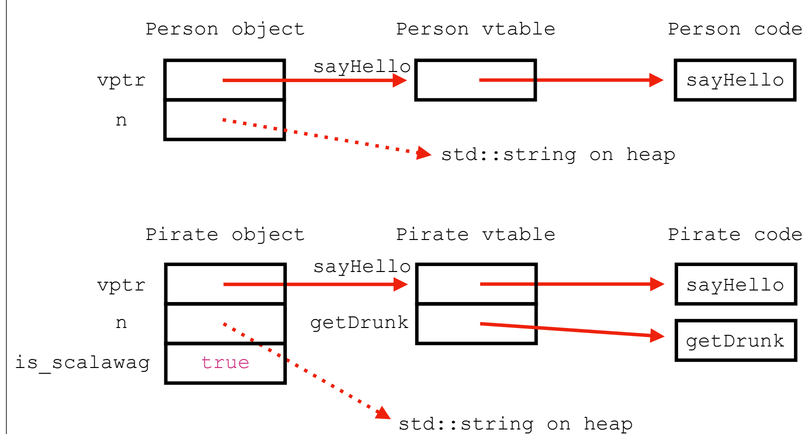
- In terms of code reuse, it makes perfect sense to implement a Stack and Queue on top a Dequeue. Dequeue has all the functionality needed.
- (Smalltalk allows one to "uninherit" methods from a superclass)
- But Stack and Queue are not subtypes of Dequeue!
- The converse is true!

```
Dequeue <: Stack
Dequeue <: Queue
```

C Features

- user-defined functions (demo)
- explicit memory functions
  - manual storage (demo)
    - malloc
    - free
    - used when memory needs to outlive activation record (example)
  - "automatic" storage (demo)
    - "local" variable; allocated on the stack
    - otherwise, allocated on the heap
    - automatically "freed" when stack popped

Virtual Dispatch



- C++ virtual dispatch does *never searches* as in SmallTalk; vtable/instance variable offsets known at compile-time.

Lambda expressions

- C++ has lambda expressions.
- They are a tad more verbose than in SML.
- Three main components.

[3] (1) { 2 }

1. Parameter list
2. Function body
3. Capture list

Templates

- C++ lets you program "generically" just like Java or SML.
- Syntax is a little different.
- Mechanism is very different.

```
        template <typename T>
        class Box {
        public:
          T x;
        }
        …
Box<double> b = new Box<double>();
b->x = 2.2;
```

Box is not "covariant"

What we want:

```
F <: Fruit
Box[F] <: Box[Fruit]
```

This is not true in Scala by default

(but the fix is simple)

```
  trait Box[+F <: Fruit] {
    def fruit: F
    def contains(aFruit: Fruit) = fruit == aFruit
  }
val box: Box[Fruit] = new Box[Apple] { def fruit = apple }
```

Implicit Conversions

Scala gives you precise control of implicit conversions.

Suppose I want to be able to write the following:

```
scala> 1.repeat(10)
```

and get

```
res4: List[Int] = List(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
```

How would I make this happen?

Declarative Programming

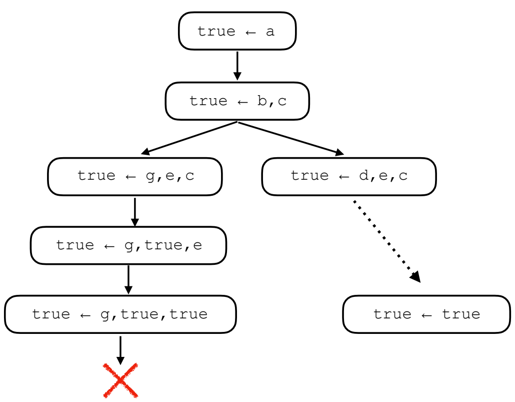
- Declarative programming is a very different style of programming than you have seen to this point.
- Mostly, you have seen imperative programs.
- In imperative-style programming, the programmer instructs the computer how to compute the desired result.
- In declarative-style programming, the computer already knows how to compute results.
- Instead, the programmer asks the computer what to compute.

Prolog

- The goal of AI is to enable a computer to answer declarative queries.
- I.e., it already knows how to answer you.
- Prolog was an attempt to solve this problem.
- Since this was early work, the input language was somewhat primitive: predicate logic.
- As you will see, formulating queries in pure logic is not the easiest thing to do.
- However, for certain classes of logic, there are known efficient, deterministic algorithms for solving every possible query.

Proof Search

- Nonetheless, Prolog is not generally sensitive to the order of the facts in a database. How does this work?
- The answer is that resolution is actually a form of backtracking search.



Domain Specific Languages

- A domain specific language (DSL) is a language designed to solve a small set of tasks.
- DSLs frequently sacrifice expressiveness in favor of ease of use.

Completeness

- A formal system is a logical system for generating formulas.
- A formal system is complete with respect to a property if all formulas having that property can be derived using the rules (axioms) of the system.

Soundness

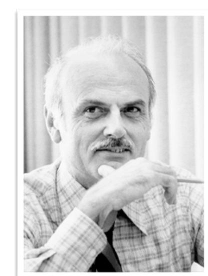
- A formal system is sound with respect to a property if all derivable formulas are true.

Incompleteness Theorem

- Kurt Gödel proved that mathematics (i.e., mathematical logic) cannot be both sound and complete wrt "provability."
- Either:
  - you can define a formal system in which you can derive all the true mathematical statements, but which also admits false statements (inconsistent), or
  - you can define a formal system in which all statements are true, but in which you cannot derive all the true mathematical statements (incomplete).
- https://youtu.be/O4ndIDcDSGc

SQL

- SQL is a DSL for querying data, invented by E. F. Codd in 1970.
- SQL limits itself to only certain kinds of queries.
- All of those queries can be answered efficiently (and by implication, they terminate).
- The language is based on a theory of data and data queries called the relational algebra.
- The relational algebra lets users efficiently query data in a form that is largely independent of the organization of the data on disk.
- This was considered a major breakthrough when it was invented.
- For many practical reasons, SQL has diverged somewhat from the relational algebra.



Relational Algebra

- The relational algebra is based on set theory.
- A relation R is a set of tuples.
  - Remember that sets contain only unique elements.
  - Also, the order of elements in a set does not matter.
- The members of a tuple are called attributes.
  - Note that the order of attributes in a tuple does not matter.
- We often think of relations as tables. But since relations are really sets of tuples, the order of attributes and rows in a table *does not matter*.
- A schema is the set of all defined relations.
- A database is a collection of instances of relations for a given schema.

Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales

Dept

DeptName	Manager
Finance	George
Sales	Harriet
Production	Charles

See the handout for more!

Have a great summer!