Introduction: The Discussion Section

Welcome to CS161!

About the Discussion Section:

- The discussion section is largely review and practice of everything that was covered during lecture in the previous week, plus some greater depth.
- This is the time when you ask questions that you didn't during lecture--leave nothing to chance!
- The general discussion format is to review topics at the high level, examine a bunch of examples that illustrate them, and then practice them by hand.

About My Role as Your TA:

- I am always available to email; don't hesitate to ask for any clarifications (though be sure to browse the Piazza forum first; your question has likely been asked already!).
- My office hours are Thursdays, 2:30 - 4:30pm in Boelter 2432. If you can't make those hours, email me to set up an appointment.

Site Features:

- You can add notes inside the website so that you can follow along and type as I say stuff! Just hit `SHIFT + N` and then click on a paragraph to add an editable note area below. NOTE: the notes you add will not persist if you close your browser, so make sure you save it to PDF when you're done taking notes! (see below)
- The site has been optimized for printing, which includes the notes that you add, above. I've added a print button to the bottom of the site, but really it just calls your printer functionality, which typically includes the export to PDF.

Piazza Forum:

I will be maintaining a class discussion board using Piazza for all of your homework and class-related questions.

Please post to the forum first before emailing me, abiding by the Forum Rules (see the sticky post on the forum):

- Register for the forum here! (http://piazza.com/ucla/winter2016/cs161)
- Access the forum here! (http://piazza.com/ucla/winter2016/cs161/home)

A Glance at AI

So you're in an AI class... what exactly is Artificial Intelligence, aside from what will ultimately replace the human race?

AI has many branches and facets, so a single definition is difficult; that said, we can say, generally, that:

> **Artificial Intelligence** is the attempt to automate the intelligent processes we often attribute to biological organisms (usually humans).

Tasks that humans perform daily are actually quite non-trivial for computers.

> What are some tasks at which humans excel over computers?

...and these are just a few applications!

Indeed, AI need not be applied, necessarily; philosophers have long debated the nature of intelligence and whether or not man has the capacity to reproduce it artificially.

We'll leave most of that philosophy stuff to north campus, though... this class will deal primarily with the technical aspects of AI.

Natural Language Processing (NLP)

Because this course dives most heavily into NLP, it's fair to take a quick look at it now.

> **Natural Language Processing** is a subfield of AI concerned with extracting meaning from text and speech.

Firstly, NLP in its entirety has not yet been solved; there are many nuances of language that computers just don't understand.

That said, history has provided us with a few main approaches to NLP, each with some strengths and weaknesses:

Approach	Description
Symbolic	[Top-down] Use known grammatical rules and structures to impose patterns on text and extract meaning by exploiting how humans typically format their text.
Statistical / Empirical	[Bottom-up] Use patterns found within text itself and try to generate rules from large corpuses of sentences.

Approach	Description
Both!	Most modern systems will incorporate sentence structure and known rules into its probabilistic analyses, combining both of the above methodologies.

NLP is also one of the oldest problems in AI, so old, in fact, that one of the fathers of computer science (Alan Turing) described a metric for our language-processing automata:

> The **Turing Test** is a game described by Alan Turing whereby human judges attempt to discern whether an unseen (text) conversational partner is a machine or another human.

The game, as Turing described it, goes something like this:

- A human judge (the interlocutor) engages in conversation between two entities that are located in separate rooms.*
- The judge spends 5 minutes (some historians argue it was 10) talking to each entity via text-message, knowing that one of the two must be a computer.
- After talking to both entities, the judge must decide which conversational partner was the human, and which was the computer.

An intelligent agent is said to have "passed" the Turing test if it fools at least 30% of its human judges into thinking that it too is human.

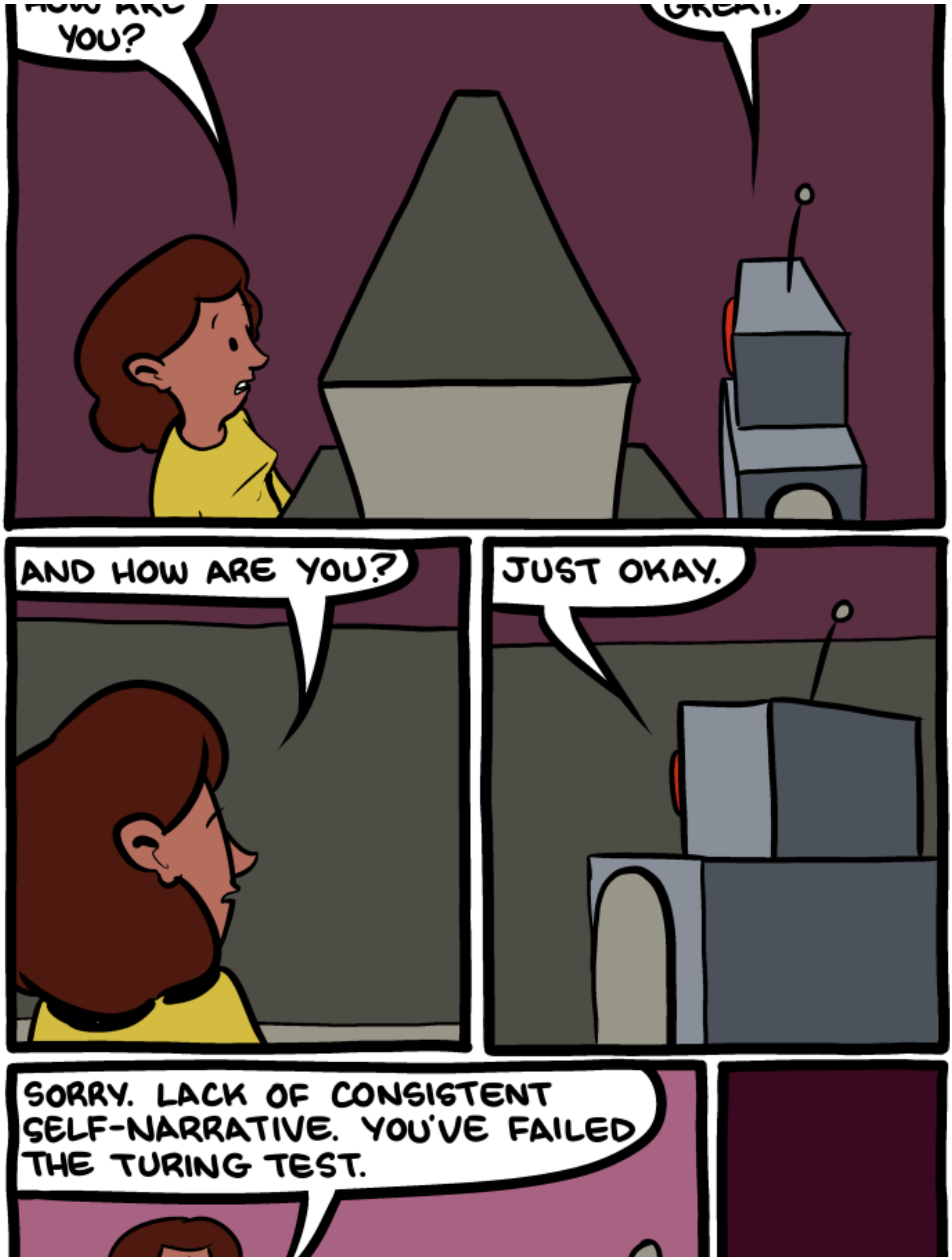
> * [Trivia] Turing, in the 1950 paper that introduced what is now considered the classic Turing test, noted that the separate rooms must be ESP-proof so that the judge could not deduce which room held the computer by means of psychic prowess (hey, it was the 50s give him a break)

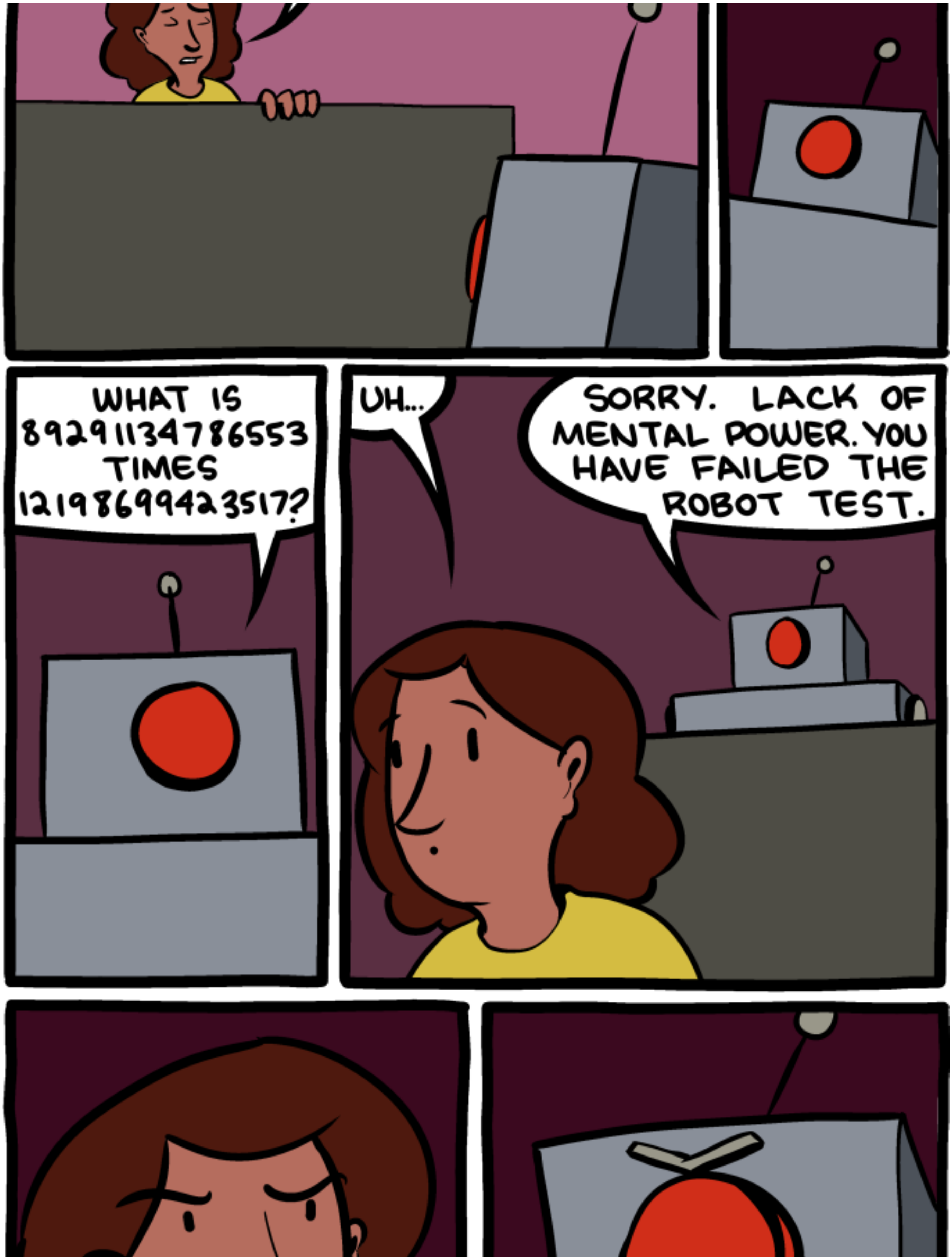
A variety of programs have claimed to have passed the Turing test, but many of these findings are dubious.

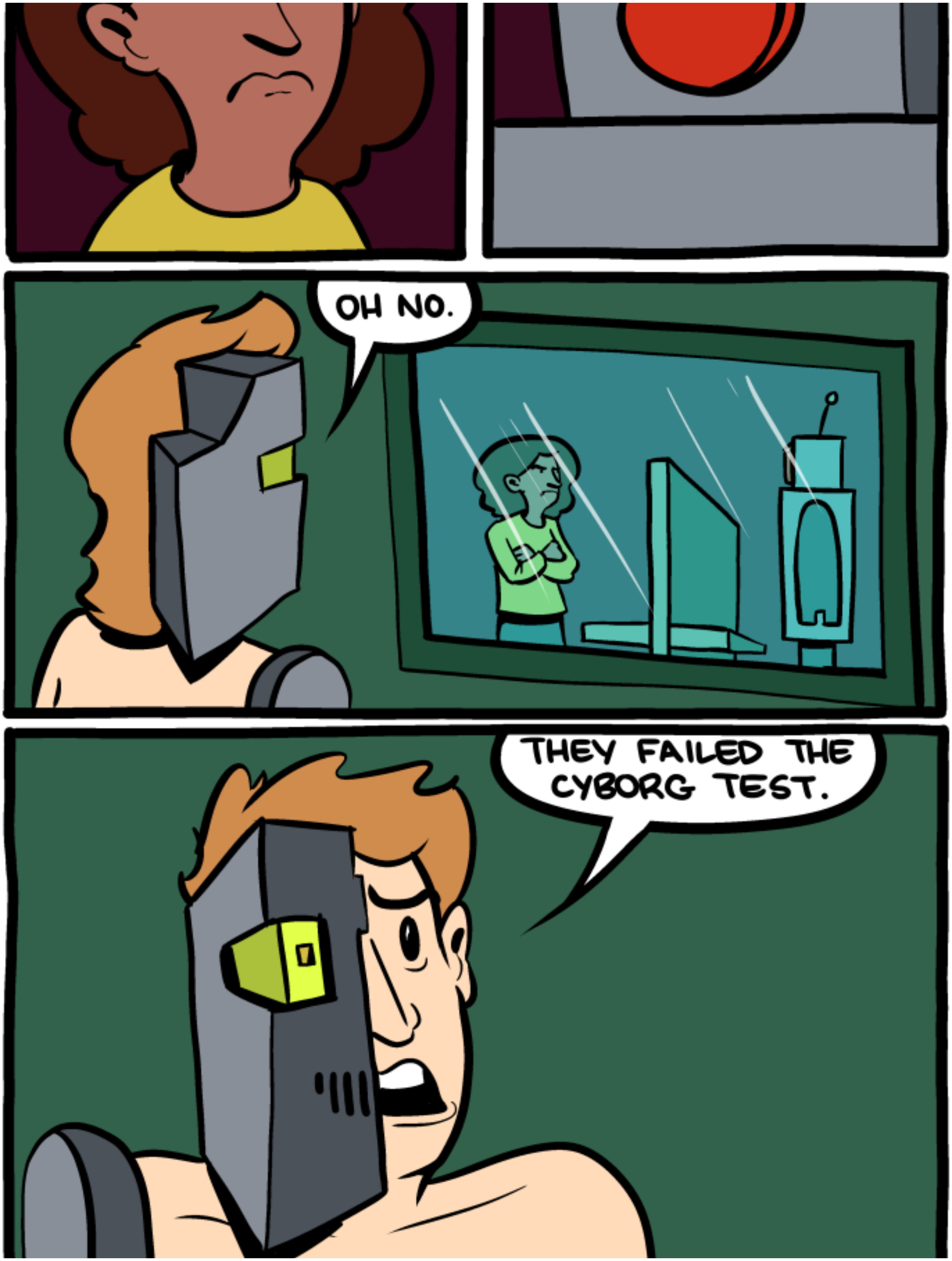
Widely regarded as the gold standard of modern Turing Test evaluation, the Loebner Prize (http://www.loebner.net/Prizef/loebner-prize.html) is an annual competition where programmers submit their chatbots to vie for the $100,000 grand prize for being "indistinguishable from a human."

The Loebner Prize competition is the formal Turing Test instantiation, and has not yet been passed (although its constraints are significantly more difficult: e.g., 25 minutes of conversation as opposed to Turing's proposed 5 minutes)

There are variants on the Turing Test as well:


YOU?
AND HOW ARE YOU?
JUST OKAY.
SORRY. LACK OF CONSISTENT SELF-NARRATIVE. YOU'VE FAILED THE TURING TEST.


WHAT IS 89291134786553 TIMES 12198699423517?
UH...
SORRY. LACK OF MENTAL POWER. YOU HAVE FAILED THE ROBOT TEST.


OH NO.
THEY FAILED THE CYBORG TEST.

Credit to SMBC

(http://www.smbc-comics.com/?id=3292#comic)

Modified Turing Tests

Modern AI evaluators have claimed that the original Turing Test has too many constraints, including, but not limited to:

- The fact that the human judge is aware that they may be interacting with a computer and take steps to fool it that would not normally follow in conversation.
- The fact that the computer is often simply being asked questions with dialogue that has been initiated by the judge (why not give the computer a chance to ask a question / respond?)
- The fact that the Turing test does not take other human communicative mechanisms into account (e.g., body language, tone, cadence, etc.)

When some of these constraints are relaxed, even very simple chatbots have completed modified Turing tests...

(Shameless plug for my own undergrad chatbot, (Gilbert & Forney, 2014))

Summary

- NLP is hard...
- ...so hard that a (version of a) test for computer intelligence proposed over 50 years ago still hasn't been satisfactorily met.
- Some, constrained, versions of the Turing test have been passed, though those studies are generally not... generalizable.
- There are a variety of different approaches to NLP each with their own strengths and weaknesses, some of which we'll discuss in this course

Then, of course, there's the end-all, be-all of Turing Tests:

Credit to XKCD

(https://xkcd.com/329/)

Feeling LISPless?

Well then let's get you set up, hmm?

⚠ This class will use the **CLISP** Common LISP interpreter, in both discussion and in grading your assignments!

As such, it is highly recommended that you use CLISP for all of your development.

⚙ Pick up a copy of CLISP here (http://www.clisp.org/)--it's cross-platform, so chances are good that it'll work on your Windows, Linux, or Mac machine!

Operating System	Installation Steps

<table>
<tr><td>Windows</td><td>At the right of this page (linked here) (http://www.clisp.org/), find a download under "our official distribution sites".</td></tr>
<tr><td>Mac OS X</td><td>

Install MacPorts using the instructions here (https://www.macports.org/install.php)
Once MacPorts is installed, open a terminal and type:
<code>sudo port install clisp</code>
You can now use your terminal to access the clisp interpreter as well as use the terminal to run your LISP code.

<hr>
Brew Alternate installation:
MacPorts not working for you? You can try the Brew tutorial located here (http://objectcoder.com/2014/01/26/installing-common-lisp-clisp-on-mac-os-x/).
(complete with strange, alien Lisp illustration)
</td></tr>
<tr><td>Linux</td><td>

On the right of this page (linked here) (http://www.clisp.org/), find your flavor of Linux and check the package listing.
Usually, you will then install the package of choice by opening a terminal and typing something like (depending on your Linux installation):
<code>sudo apt-get install clisp</code>
You can now use your terminal to access the clisp interpreter as well as use the terminal to run your LISP code.

</td></tr>
</table>

If you're having any issues installing CLISP, let me know and I'll do my best to help!

Other Tutorials

Want more than this dinky guide can provide? Here are some really complete LISP tutorials, only some material of which we'll be using in this course:

- CMU's LISP Text (https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node1.html)
- Practical Common LISP (http://www.gigamonkeys.com/book/)

An Introduction to LISP

For this course, we'll be using the LISP programming language.

LISP is a LISt Processing programming language, often mocked for its "Lots of Insipid, Stupid Parentheses"

Why LISP for AI?

- LISP syntax is tree-oriented (and therefore, recursion oriented), which is useful for NLP and many other tree-based operations
- Because of its tree-like properties, making proofs and algorithmic analysis is simplified
- LISP boasts interchangability of data and program code, which allows for ease of self-modifying behavior

There are a variety of different LISP implementations. This class, and guide, describes the Common LISP variant.

Atom & Eval

Get it? Because we're starting with the basics? :/

Atoms are simple data types in LISP such as numbers and strings. These are indivisible into smaller components, thus the name.

Symbols are atoms that possess a name, possibly a value, and possibly a function definition.

In this way, all symbols are atoms, but not all atoms are necessarily symbols.

NOTE: Symbols are case-insensitive, which means that a symbol representing some function called SYMBOL-ENOUGH is equivalent to referencing the same function SyMbOl-EnOuGh

If a symbol is defined, then it can be evaluated (like a function name used to evaluate its definition), otherwise, you'll get an error.

A **list** is a sequence of atoms and other (recursively defined) lists.

Because everything is a list in LISP, that means we can have two different list interpretations:

<table>
<tr><th>Interpretation</th><th>Syntax</th></tr>
<tr><td>Lists as code</td><td>Almost all LISP expressions will match the following format:

<code>(function-name argument1 argument2 ... argumentN)</code>

Where "function-name" is a symbol designating a function definition, followed by a space-separated list of its arguments.</td></tr>
<tr><td>Lists as data</td><td>If you would like a list of data elements, you simply prepend a list using a single quote:

<code>'(item1 item2 ... itemN)</code>

...which is actually a read-macro expansion for:

<code>(quote (item1 item2 ... itemN))</code>

...but we'll use the single-quote short-hand to simplify</td></tr>
</table>

All lists are in the form of Polish Prefix notation, and are evaluated from left to right; atoms evaluate to themselves.

Basic Arithmetic Operators

The following symbols serve as LISP's basic arithmetic operators:

```
;; Addition
(+ arg1 arg2 ...)

;; Subtraction
(- arg1 arg2 ...)

;; Multiplication
(* arg1 arg2 ...)

;; Floating-point Division
;; Returns a ratio when arguments are ints
(/ arg1 arg2 ...)

;; Modulus
(mod arg1 arg2 ...)

;; Floor / Ceiling / Round
(floor arg1)
(ceiling arg1)
(round arg1)
```

Example

Let's start off simple; what will the following LISP expressions evaluate to?

```
;; #1
(+ 1 2 3)

;; #2
(* 1 (- 1 2 1) 3)

;; #3
(* (/ 1 2) (/ 1.0 2))

;; #4
(* (/ 1 2) (/ 1 2))
```

Now, remember we said that LISP treats lists as code and data differently? Well there's a conversion!

The **eval** function converts a data list into a code list.

Example
What will the following code print out?

```
;; #1
(mod (eval '(* 1 2 3)) (- 3 1))

;; #2
'(* 1 2 3)

;; #3
'(eval '(* 1 2 3))
```

Boolean Logic

In terms of LISP boolean logic, there are two atomic values: `nil` (false) and `t` (true)

ANYTHING non-nil is considered true, except for the empty list `()` which is also considered nil.

Here are some basic boolean operators at your disposal:

```
;; Negation (Unary)
(not arg)

;; Logical AND
(and arg1 arg2 ... argN)

;; Logical OR
(or arg1 arg2 ... argN)
```

There are some peculiarities with AND and OR:

- If a logical AND returns true, then it will return the LAST true element of its argument list.
- If a logical OR returns true, then it will return the FIRST true element of its argument list.

Example
What will the following code print out?

```
;; #1
(not (mod 2 2))

;; #2
(and (* 2 2) t nil)

;; #3
(and (* 2 2) t 3)

;; #4
(or (* 2 2) t nil)
```

Numerical Comparators

Everyone's favorite numerical comparators return in LISP:

```
;; Numerical equivalence
(= n1 n2 ...)

;; Less than
(< n1 n2 ...)

;; Less than or equal
(<= n1 n2 ...)

;; Greater than
(> n1 n2 ...)

;; Greater than or equal
(>= n1 n2 ...)
```

NOTE: You can provide more than 2 arguments to each comparator, in which case an expression like:

```
(> n1 n2 n3)
```

...will return true if and only if n1 > n2 > n3

Example

What will the following code print out?

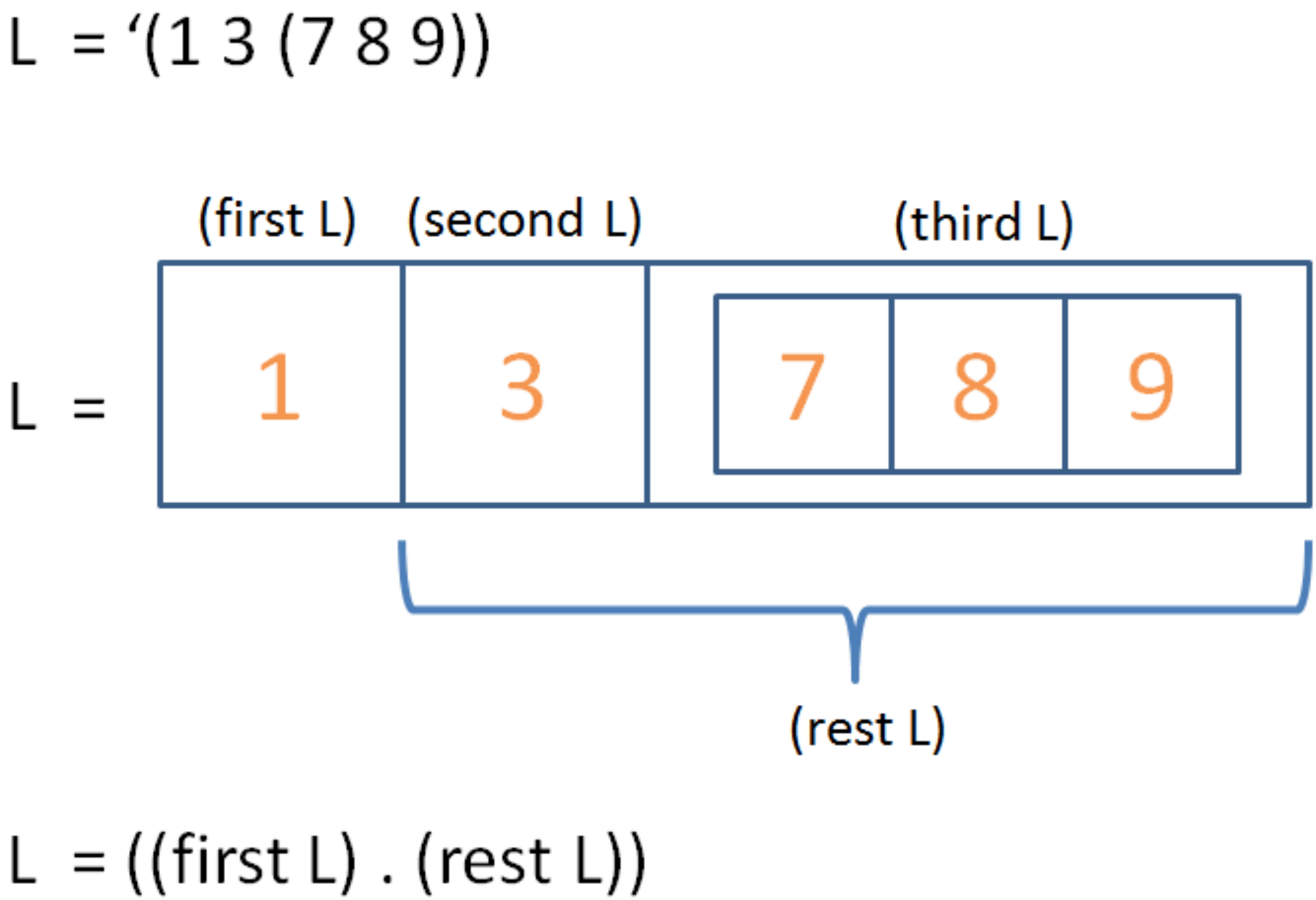
```
;; #1
(>= (+ 1 2) 3 4)

;; #2
(= 1 1.0)

;; #3
(= 0.25 1/4)
```

List Accessors

As we know, lists in LISP are just as you'd expect: each cell of a list can contain an atom, or another list!

Example

Observe the depiction of the following list's cells:



⚙ You can use **list element accessors** to get the contents of a list's cells; these include:

```
;; Get the first element of list L
(first L)

;; Get the second element of list L
(second L)

;; Get a *sublist* of the last
;; element of list L
(last L)

;; Get the nth element of list L
;; (indexed starting at 0)
(nth n L)

;; Get the sublist in L composed of all
;; elements except for the first
(rest L)

;; Get the sublist of L composed of all
;; elements except for the first n (count
;; of n, not the index)
(nthcdr n L)
```

Example

What will the following code print out?

```
;; #1
(first '(1 2 3 4))

;; #2
(rest '("this" "example" "rocks"))

;; #3
(nthcdr 2 '(2 4 6 8))

;; #4
(nth 2 '(2 4 (6 7) 8))
```

List Manipulation

So we've looked at how to get things out of lists, but how about constructing them from different atoms and lists?

> ✿ The `cons` function returns a list composed of the first element (exactly) with the elements of the 2nd argument "tacked on" after.

Generally, we use cons to prepend an atom to the front of a list. For example:

```
;; Any list L is equivalent to:
(cons (first L) (rest L))
```

This one's a bit tricky; let's examine the outcomes of its different arguments:

Using `cons`	**Second Argument: Atom**	**Second Argument: List**

First Argument: Atom	`;; OK, but might not`
	`;; be what you want...`
	`(cons 1 2)`
	
	`;; (1 . 2)`
		`(cons 5 '(6 7))`
		
		`;; (5 6 7)`

First Argument: List	`;; OK, but might not`
	`;; be what you want...`
	`(cons '(1 2) 3)`
	
	`;; ((1 2) . 3)`
		`(cons '(1 2) '(3 4))`
		
		`;; ((1 2) 3 4)`

So what does the period mean when you see it in a list?

It is essentially a symbol representing the cons operation. For example:

```
'(1 2 3 4)

;; is equivalent to:
'(1 . (2 3 4))

;; also equivalent to:
'(1 . (2 . (3 4)))

;; also equivalent to:
(cons 1 (cons 2 '(3 4)))
```

Be wary of this symbol in your code! It probably means you tried to cons something that you didn't mean to... instead, you'll probably want one of the operations discussed in the next section.

One final note, observe the following special behavior of cons involving nil:

```
;; #1
(cons nil 5)

;; #2
(cons 5 nil)
```

The `list` operation creates a new list with all i arguments as the ith elements of that new list.

Example

What does the following code print out?

```
;; #1
(list 1 2 3)

;; #2
(list 1 '(2 3))

;; #3
(list (cons 1 '(2 3)) (cons 4 '(5)))
```

The `append` function takes the elements of each argument list and embeds them at the end of the previous argument list.

What does the following code print out?

```
;; #1
(append '(1 2 3) '(4 5 6))

;; #2
(append (list 1 2 3) (cons 4 '(5 6)))

;; #3
(append (list 1 2 3) '(4 5) '((6 7)) )
```

NOTE: Appending nil to a list will not change that list. Very useful!

Utility Functions

You can use the `length` function to determine the size of a list.

You can use the `null` function to determine if a list is the empty list.

Example
What will the following code print out?

```
;; #1
(length '(1 2 3))

;; #2
(length '(1 (2 3)))

;; #3
(length '(()) )

;; #4
(null '())

;; #5
(null '(()) )
```

Predicates

A **predicate** is a function that denotes some property of an atom or list, including:

```
;; Returns true if N is odd
(oddp n)

;; Returns true if N is even
(evenp n)

;; Returns true if x is a number
(numberp x)

;; Returns true if x is a string
(stringp x)

;; Returns true if x is a symbol
(symbolp x)

;; Returns true if x is a list
(listp x)

;; Returns true if x is an atom
(atom x)
```

Example

What will the following code print out?

```
;; #1
(oddp 1)

;; #2
(numberp 5.32)

;; #3
(stringp "Test")

;; #4
(stringp 'Test)

;; #5
(symbolp 'Test)

;; #6
(symbolp "Test")

;; #7
(listp '(1 2 3))

;; #8
(atom '(1 2 3))
```

The `equal` function determines if two objects are equivalent in value and type.

WARNING: the `equal` and `=` operators are, ironically, NOT EQUIVALENT.

`equal` can operate on non-numeric quantities and compares type as well as value.

Comparisons are made between atoms and lists slightly differently:

- Atoms X and Y are equal if they are objects of the same type and value
- Lists X and Y are equal if they are contain the same number of elements, and each element i is also equal between each list

Example
Compare the following outcomes:

```
;; #1 - Remember, value AND type
(equal 1.0 1)
(= 1.0 1)

;; #2
(equal 'x 'X)
(equal "x" "X")

;; #3
(equal '(1 2 3) (list 1 2 3))
```

LISP Variables, Control Flow, and Functions

Are we having fun yet?

Let's look at some more advanced LISP stuff; this is where it really gets interesting...

Variables

LISP **variables** are just like any other in your programming language of choice, except they can be used to represent any data type dynamically.

There are two primary means of instantiating variables:

Function	Description

setq	The `setq` operation is a global binding of a symbol to some definition.
	
	⚠ You should almost never use this for your functions in class, or really ever outside of a testing framework! It is evil!
	
	`;; Syntax:`
	`(setq sym1 def1 ... symN defN)`
	`;; Returns defN`

let (or let*)	The `let (or let*)` operations provide local variable definitions, and are only defined in their scope.
	
	These are what you shall use almost exclusively in your work.
	
	`;; Syntax:`
	`(let* ((sym1 def1) (sym2 def2) ... (symN defN))`
	`  expression1`
	`  ...`
	`  expressionM`
	`)`
	`;; Returns value of expressionM`

Let's look at some examples of each:

Example
What will the following code print out?

```
(setq
  lame 5
  example '(1 2 3)
)

;; NOTE: setq returns the value
;; of the last instantiated variable

(+ lame (first example))
```

Again, unless otherwise specified, you should avoid using setq in your assignments; you may use it for test code!

⚙ The `boundp` predicate tests whether or not a given symbol is bound to some definition.

```
;; Assuming no other code above, is 'TEST bound yet?
(print (boundp 'TEST))

(setq TEST 5)

;; Now is 'TEST bound?
(print (boundp 'TEST))
```

[HINT] You may find boundp useful in your homework for determining if gaps are conatms!

We'll talk about this later...

Now, I know what your next question is: What's the difference between `let` and `let*`?

- `let` performs variable instantiation **in parallel (at the same time)** so if there's any dependence between instantiations, you'll get an error
- `let*` performs variable instantiation **in sequence (one after the next)**, useful for dependence between instantiations.

Example
Observe the difference between the two below:

```
;; [X] ERROR: Will not work because
;; Y relies on X and the let operation
;; instantiates in parallel
(let ( (X 5) (Y (+ X 5)) )
  ;; ...
)

;; [!] OK: Will work because
;; Y relies on X and the let* operation
;; instantiates X first, and then Y
(let* ( (X 5) (Y (+ X 5)) )
  ;; ...
)
```

ℹ Both versions of `let` will return the last expression mentioned in the block.

Example
What will the following code print out?

```
(let* ( (M 20) (N (+ M M)) )
  (+ N M)
  (- N M)
)
```

Control

Because we'll be using recursion almost exclusively in our work, we'll need some means of performing different actions based on different states.

We use the following control syntax to decide what to do based on the state of our function inputs:

The `cond` (conditional) operator performs the code block of its first non-nil case.

The syntax of a `cond` statement is as follows:

```
(cond
  ((case1) expr1_1 expr1_2 ...)
  ((case2) expr2_1 expr2_2 ...)
  ...
  ((caseN) exprN_1 exprN_2 ...)
)
```

The above says, "Starting at case1, keep trying to find a case that returns t, and if so, execute those expressions. If no case returns t, then return nil at the end of the cond."

Example
What will the following code print out?

```
(let* ( (X 5) (Y '(X 6 7)) )
  (cond
    ; Case 1:
    ((> X (nth 1 Y)) '(1) )

    ; Case 2:
    ((< X (second Y)) '(2) )
  )
)
```

Functions

Now that we have variables and control flow, it's time to get down to functions...

Here is the canonical prototye for a function definition:

```
;; Top level function comment
;; (I like 2 semicolons)
(defun function-name (param1 param2 ... paramN)
  ; Local variable definitions
  (let* (
    (local1 def1)
    (local2 def2) )
    ...
    (cond
      ; Inline case explanation 1
      ((case1) exp1 ...)

      ; Inline case explanation 1
      ((case2) exp2 ...)

      ...
    )
  )
)
```

A function returns the value of the last expression that was evaluated.

Example
What will the following code print out?

```
;; [Purpose]
;;   Returns the value of n raised
;;   to integer power pow
;; [Inputs]
;;   n (number): number to raise to power pow
;;   pow (integer): power to raise n to
;; [Output]
;;   (number) n ^ pow
(defun power (n pow)
  (cond
    ; Base case: pow is 0, return 1
    ((= pow 0) 1)

    ; If power is positive, we recursively
    ; multiply our result by n
    ((> pow 0) (* n (power n (- pow 1))))

    ; Otherwise, power is negative, so we have
    ; to divide 1 by our result
    ((< pow 0) (/ 1 (power n (* pow -1))))
  )
)

; We can run some tests here:
(power 10 -2)
(power 10 -1)
(power 10  0)
(power 10  1)
(power 10  2)
```

Style

A few quick notes about style and indentation:

- ALWAYS provide top-level function comments that describe the function's (1) purpose, (2) inputs, and (3) expected outputs!
- You should comment your base and recursive cases for when they are matched and how they respond.
- Always indent nested code! This includes, but is not limited to:
  - Function bodies indented from the `defun` operator.
  - Bodies of the `let, let*` operators.
  - Conditional cases indented from the `cond` operator.

Practice

Try your hand at a few practice problems!

Example

Complete the code skeleton described below:

```
;; [Purpose]
;;   Returns a list that is the reverse of input
;; [Inputs]
;;   L (list) to reverse
;; [Outputs]
;;   (list) L reversed
(defun reverse-list (L)
  (cond
    ; Base case: L is nil, so return nil
    ((null L) nil)

    ; Recursive case: append the first element to the rest
    ; of the list called recursively
    ; [!] Watch out: remember what append expects!
    (t (append (reverse-list ( ??? )) (??? ( ??? ))))
  )
)
```

Example

Complete the code skeleton described below:

```
;; [Purpose]
;;   Replaces the nth element of the given list L
;;   with the given list R
;; [Inputs]
;;   L (list) to replace element within
;;   n (int) index to replace in L
;;   R (atom / list) replace L[n] with R
;; [Outputs]
;;   (list) L with replacement R
(defun replace-element (L n R)
  (cond
    ; Base case: L is empty, so return nil
    ( ??? )

    ; Base case: n has reached 0, so we are
    ; at the index in L to place R
    ((= n 0) ( ??? ))

    ; Recursive case: L is not empty and n is not 0, so
    ; add the first element of L to our result, and then
    ; recurse on the rest of L and n - 1
    (t (??? (first L) (replace-element ( ??? ) ( ??? ) R)))
  )
)
```

LISP Typical Workflow

There are a few tips and tricks using CLISP to help with your testing and development.

I assume, in this section of the guide, that you are working from the CLISP command line / terminal interface.

Your workflow for assignments will go something like this:

- Use whatever IDE or text editor you'd like for developing with LISP, and you will save files with .lsp extension.
- Perhaps you have a file with your functions called "funcs.lsp". If you want to test your functions in another test file, you can load them using:
  - `(load "source.lsp")`
  - `(load "source.lsp" :compiling t)` - [optional] This CLISP specific command will compile your code before running, which might catch more bugs
- Within your interpreter, you can change directory to the location of your LISP files:
  - `(cd)` - Lists your current directory

- `(cd "directory")` - Changes to the given directory

- You can then run any of your files in the current directory by typing:
  - `clisp "source.lsp"` - Runs your code
  - `clisp -c "source.lsp"` - [optional] Compiles your code for more robust error checking
  - `clisp -i "source.lsp"` - Loads your code and then enters into the interpreter

Debugging

Here are a couple of helpful functions you might find useful during debugging!

✿ The `trace` function will print out a trace of each recursive call to the provided function name.

```
; Begins tracing the call stack of
; the power function
(trace power)

(power 10 1)
(power 10 2)

; Stops tracing the call stack of
; the power function
(untrace power)

; Alternately:
(untrace)
; Will stop tracing all functions
```

✿ Use the `print` function to return the value of its argument, as well as print that argument to the console.

✿ Use the `format` function to print a formatted string with all instances of ~S replaced, in order, by its arguments (see example below)

```
; Example for print
(let* ( (X 5) (Y (* X X)) )
  (print (+ (print X) X))
)

; Example for format
(let* ( (X 5) (Y (* X X)) )
  (format t "The value of X is ~S and Y is ~S" X Y)
)
```

(Thanks to Evan Lloyd for the debugger info and outline!)

Homework 1

Homework 1 asks you to define some functions relevant to frames.

> **Frames** are data structures in LISP that attempt to break a natural language sentence down into its constituent, semantic components.

What does that mean?

Well, we're interested in picking apart a text sentence, with all of the nuances of human writing, into computer-friendly pieces for semantic processing (extracting meaning).

A frame abides by the following syntax:

```
frame      -> (pred slotfiller*) | NIL
pred       -> atom
slotfiller -> slot filler
slot       -> atom
filler     -> frame | gap
gap        -> variable | atom
variable   -> (V atom)
```

Here's an example:

Example
A frame representing the sentence:

"Andrew taught class."

```
(ACT AGENT (HUMAN F-NAME (ANDREW)
                  GENDER (MALE))
     ACTION (TAUGHT)
     OBJECT (CLASS))
```

Some things to note:

- Note the syntax structure with outermost predicate ACT, which has 3 slots: AGENT, ACTION, OBJECT.
- The AGENT slot actually has a frame as a filler with predicate HUMAN and slots F-NAME and GENDER.
- In Lisp, these are all lists of symbols, with the nesting schema specified by the above syntax.

Using what we've learned today about Lisp, you should be able to complete all of your HW1 problems.

If there's time, we can go over a few of those now.

That's it for today, look over the functions you've been assigned and let me know if any need clarification! Good luck!
