SYMBOLIC DATA

More than just numbers

WELCOME BACK (SORT OF)

TAKING CARE OF YOURSELF

- You didn't come here for my personal life advice, but you're stuck with it anyhow
- (Some of these elements may be challenging for you)
  - Keep a normal routine!
  - Keep hydrated (this is really tough for those of you who are used to bringing a water bottle everywhere and rely on motion through the buildings to remind you to drink)
  - Stay mentally engaged with something
  - Try to separate work spaces from living spaces
  - Keep in touch with your friends via video tools etc

ONE-ON-ONE APPOINTMENTS

- I would like to have 5 minutes of your time once per week
- I honestly can't afford much more than that for a class this size, with all of my other responsibilities
- We'll use a scheduled meeting inside brightspace
- This is a learning experience for me, too!

- https://www.picktime.com/UofDMirabelli

- 8-10pm CST Tuesday & Thursday
  - Other times as requested

MORE TO LIFE THAN NUMBERS

(seriously)

THERE'S MORE TO LIFE

- Numbers aren't enough to provide all of the information we need when storing data for our programs
- We often have a need to manipulate *symbols* as well as numbers in order to give context to our programs

QUOTATION

- Forming compound data with only numbers makes it difficult to assign any further meaning to those numbers
- Using quotation can provide deeper meaning to our numbers
  - (define speeds (4 20 100))
  - What does each individual value mean?

A DEFINITION WITH MEANING

```
((foot 4)
 (bicycle 20)
 (car 100))
```

* Now these numbers mean something to the casual reader

* We can still compose a list of the numeric value by mapping the cdr of each element

THE PROBLEM OF VARIABLES

We also have the ability to define a variable:

```
(define a 4)
(define b 5)
(list a b)
```

* the list has values of (4 5), not (a b) as literals

* Sometimes we have a desire to have literals as well as variables, and even with the same name

* The single quote allows us to have a literal value

(LIST A B)

```
(define a 4)
(define b 5)
(list a b)          ⟹ (4 5)
(list 'a b)         ⟹ (a 5)
(list 'a 'b)        ⟹ (a b)
(list (a 'b)        ⟹ (4 b)
```

A single quote character quotes the next object

LISTS OF SYMBOLS

- These symbols are not limited to single characters
- (define c 'foo)
  - A full word can be in place as the definition
- (define d '(a b c))
  - A list of single character values can be held as the value in such a definition

COMPARING SYMBOLS

- We can compare symbols for equality by using the *eq?* operator
- (eq? a 4)
- (eq? b 5)
- (eq? b 'b)
- (eq? 'foo c)

- In each case, the values of each are compared, not the memory location — memory locations are irrelevant for this

CREATING A LIST CHECKER

```
(define (memq item x)
  (cond ((null? x) false)
    ((eq? item(car x)) x)
     (else (memq item (cdr x))))))
```

TESTING THE MEMORY CHECKER

```
(memq 5 (list 4 5 6))

(memq 'apple '(pear banana prune))

(memq 'apple '(x (apple sauce)))

(memq 'apple '(x (apple sauce) apple peach)))
```

SETS

SETS

- Sets are, of course, a really cool type of data structure
- Collection of *distinct* objects
- Implementation is not necessarily obvious
- Functionality *is* fairly obvious, though
- Use "wishful thinking" (or software contracts) to design a set

UNION-SET

- *union-set* would take two sets as its formal parameters, and from those two sets, it would return a single set
- The set returned would contain all elements from both of the two original sets

INTERSECTION-SET

- The intersection of sets takes two sets as parameters, and returns a set as its result
- The resultant set contains the elements that are common to both sets

ELEMENT-OF-SET?

- We use element-of-set? to determine if a set contains the particular element being checked
- This takes a single element and a set as its formal parameters, and returns a boolean value if the element is within the set
- This method is one of the two keys to composing a set

ADJOIN-SET

- *adjoin-set* also takes a single element and a set as its formal parameters, but it returns a set containing all elements originally in the set, as well as the new element
- How should this behave if the provided element is already within the set?

SETS AS A LIST

We can use memq earlier as a pattern for developing set ownership

Rather than using eq?, we will use equal? so that the set elements need not be symbols

```
(define (element-of-set? x set)
  (cond ((null? set) false)
    ((equal? x (car set)) true)
    (else (element-of-set? x (cdr set)))))
```

ELEMENT-OF-SET?

```
(define (element-of-set? x set)
  (cond ((null? set) false)
    ((equal? x (car set)) true)
    (else (element-of-set? x (cdr set)))))
```

```
(element-of-set? 4 (list 2 3 4))
```

IF WE HAVE ELEMENT-OF-SET

- When we have a large problem to solve, such as "how might we implement a set", the key is to find as small a part of the problem as we can first
- First, we define an interface
  - What are the actions that might need to be taken on a set?
  - We might to look at set theory for an answer to this

DIVERSION

WHY DO WE DO THIS?

- Let's be real here
- "Set" exists in every major and modern programming language
  - I encourage you to prove me wrong (experimental languages do not count)
- Standard libraries are very large

IT'S NOT JUST SETS

- This semester we've been dissecting every major component of a programming language, including such simple things as addition
- Why do we do this?

COMPUTER SCIENCE IS NOT TRADE SCHOOL

- If we wanted to get you 'up to speed' as quickly as possible, ready to take on a lucrative job in the software industry, there are far easier ways to do this!
  - Bootcamps
  - Sketchy YouTube video series
  - RTFM
  - And so on
- Instead, we want you to understand how these tools work at their most core level
- This sets you up for a future of more learning

PLEASE BE PATIENT

- So be patient if it seems we're covering some territory that's irrelevant or at least not as "useful" as manipulating a package manager for JavaScript
- Slow development of critical reasoning about computer programs is essential
  - Emphasis on "slow"

PROGRAMMING IS LIKE BASKETBALL

- OK, it isn't at all 😜
- However, one of my favorite things to ponder is how the very basics of programming works
- Take Michael Jackson, LeBron James, or whoever your favorite athlete is
  - How much time do you think they spend on *fundamentals*
  - It's constant re-evaluation of what they know

FUNDAMENTALS

- So please, find time to focus on the fundamentals of programming
- Slow down
- Think hard about the basics. This can be hard to do if you "just know" how computers work
  - But do you?
  - I mean really?
- So slow down

/ DIVERSION

BUILDING ADJOIN-SET

- We can build *adjoin-set* once we know that we have *element-of-set*
- If an element exists within the set, return the existing set. Otherwise, append the element to the end of the set
- This gives us a set implementation as an *unordered list*
- Most modern languages implement sets in an *unordered* fashion

ADJOIN-SET

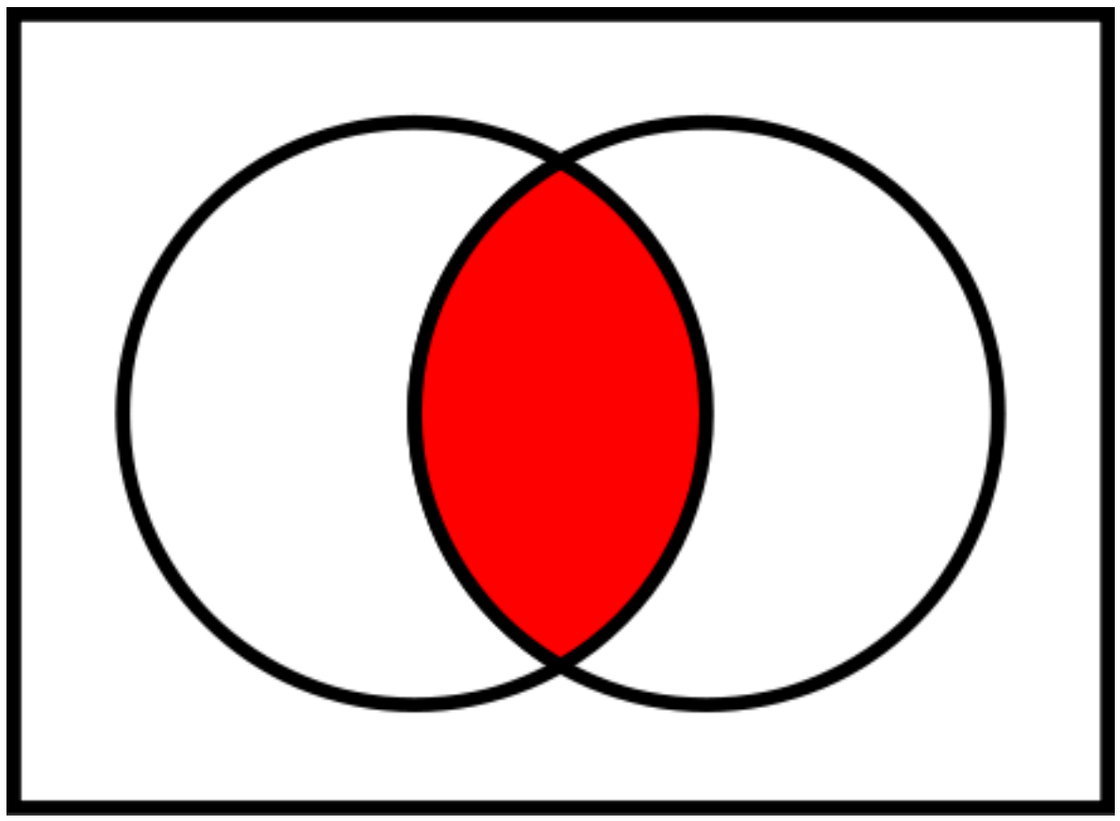
```
(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set)))
```

This is almost preposterously simple

SUBSTITUTION

INTERSECTION-SET

- We need to come up with a strategy for implementing intersection-set
- intersection-set returns a set where the elements are part of both input sets



RECURSIVE STRATEGY

- Assume that we've already solved the problem of providing the intersection of *set2* and the *cdr* of *set 1*
  - This is entirely reasonable. Why?
  - If we're trying to say we can provide an intersection of two lists (which are each assumed to be sets), we can say two more things
    - Each list has a *car* and a *cdr*
    - The *cdr* of a list is itself a list (and in this case a set)
- Therefore we can make the assumption for this strategy

RECURSIVE STRATEGY

- Therefore, assuming we can provide an intersection of the *cdr* of *set 1* and *set 2*, then all that remains is to determine if we should add the *car* of *set 1* to the intersection.
  - We solve for the simplest case—the intersection of a single element and a full set
  - Then this simple solution is propagated through the use of recursion
  - *trust the recursion*
    - (I think I've said this before)

INTERSECTION-SET

```
(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2) '())
        ((element-of-set? (car set1) set2)
          (cons (car set1)
                (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set)))))


; tip: break this apart by parenthesis
```

ALGORITHM EFFICIENCY

- Theoretically, one should be concerned about the efficiency of one's algorithms
  - This is very difficult to do on modern computer systems
  - (being concerned, not being efficient)
- There's always CPU and memory to spare right?
  - Embedded systems
  - Threads / Distributed computing
  - Tight loops
- So, yes it matters… sometimes

HOW EFFICIENT IS INTERSECTION-SET?

- The efficiency of *intersection-set* is *entirely* dependent upon the efficiency of *element-of-set?*
  - Each iteration through *set 1* requires that we potentially iterate through *all members* of *set 2* using *element-of-set?*
    - The worst case is that the element does not appear in *set 2*
  - Therefore, if the set has n elements, *element-of-set?* might take up to n steps $\Theta(n)$
  - *adjoin-set*, which also uses *element-of-set?* , also grows in the same way. $\Theta(n)$
  - *intersection-set* grows with the sizes of both sets, or $\Theta(n^2)$
  - So will *union-set*

A PERSONAL CHALLENGE

- Challenge yourself to this: *how can I implement union-set*?
  - Note: this is not *homework*, but should be considered an important part of this class session
  - Work that brain!
  - This will have much in common with *intersection-set*
  - We will review next class day