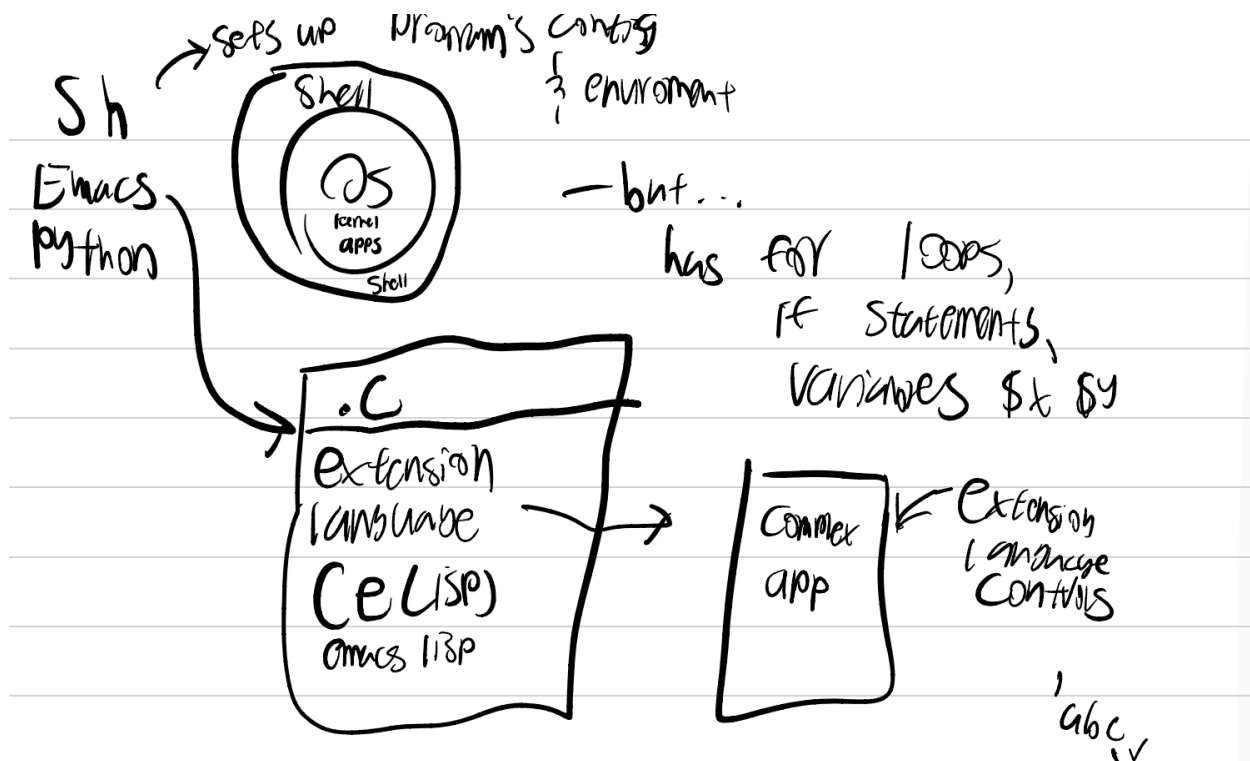




Week 3 Wednesday

Elisp

- Two approaches: the Shell is used for configuring the real stuff, vs. the Shell is used for doing real computation with loops and stuff.
- We've been looking at an example of a scripting language, the shell, in which the basic model works as follows: you have an OS. Around it, in a very thin layer, you have the shell. The way the shell gets things done isn't by doing anything itself — the shell is just a little configuration language that you use to set up the programs you really want to run. The OS is a kernel and has a bunch of apps. Your goal in the shell is to run those applications in the appropriate environment, where the standard output of one is hooked up to the standard input of another, etc. Each app can run without worrying how it was invoked. The shell is in charge. Except a lot of times with the shell you start up one app and run it kind of forever, and the shell gets out of your way. That's one way to think about software construction. This approach basically says that the shell is for configuring the real stuff.



- But the above is a bit of an exaggeration because you can do real computation in the shell. e.g. the shell has for loops. e.g. `for i in *.c, do rm "$i"` (not recommended because you're basically deleting your source code). You can write for loops, while loops, functions, and all of the stuff you're used to in CS 31. But that's not how most people actually use the shell. They use it to do other things. They may have a if statement that runs a command. They'll set x to be abc, otherwise set y to def ghi, etc. until eventually passing x and y to some other command. So the focus in the shell is how to start up these commands in the right way.

```
if cmd arg3
then x="abc"
else y="def ghi"
fi
cmd $x $y
```

- Now we'll talk about a different way of hooking stuff together. The shell is intended to be a little thing. It's a just a glue pot that lets you glue together other stuff. It's not intended to be anything more.

- Emacs works another way. Not just Emacs, lots of other systems work this way. But with Emacs, the goal is, in some sense, bigger.
- Emacs source code is divided into two parts: 1) a small part written in .c (source code looks pretty familiar if you're used to C++, but they have some macros) and 2) most of emacs is written in an extension language which has the following feel about it: you have an app that's complicated and does a lot of things. You want to control how it runs. If it's a really simple app, like `tr`, it may just have 10 options which you can quickly learn and be done. But a more complicated app might have hundreds or thousands of options. It's just too complicated to think of it in the option way. In this complex app, they'll support configuring the application, not by options, but by having an extra little programming language that you can use to run and affect programs inside the app, except the program is written just for that app. You can write code in this extension language and feed that into the complex app in order to tell it what to do. Can think of it as learning another programming language just designed for the application. It's a very common approach in many applications.
- `tr -c, tr -s` is an example of a simple language that wouldn't need a separate little programming language; the option flags are sufficient.
- JavaScript started out as an extension language for a browser. SQL started out as an extension language for database systems. (Q: how are JavaScript and SQL extension languages?)
 - An extension language is a language designed to be embedded inside another system to extend its functionality, rather than functioning as a standalone programming environment. Both JavaScript and SQL fit this definition in their original contexts.
 - JavaScript was not originally a standalone language. It was designed to extend web browsers (specifically Netscape Navigator in 1995). HTML and CSS were static, but JavaScript allowed dynamic content, interactivity, and scripting
 - SQL (Structured Query Language) was developed to interface with relational databases. It extends a database system by providing a declarative way to query, insert, update, and delete data. SQL itself does

not run as a standalone program; it needs a database engine like MySQL, PostgreSQL, or SQLite.

- Emacs' extension language is **Elisp** (short for Emacs Lisp). It's a variation of Lisp, one of the oldest programming languages still in widespread use. elisp is designed to be an extension language for emacs. One way to think about elisp is: elisp is ordinary lisp that's been altered/"extended" to have a bunch of primitives to deal with the stuff that emacs wants to deal with. e.g. it's an IDE, so you want to be able to do text editing, compile programs, build programs, etc. Elisp can let you do all that. At its core, it's a lisp language.
- Most of emacs is written in elisp! It started off mostly in C actually, but people kept extending and extending it, and it just became so much easier to extend it in lisp than C (C and C++ are kind of languages intended for hardcore engineers and low-level programmers, which are a minority in the software development world as most people don't want to program at that low level and want to think more abstractly).
- In order to get stuff done in emacs, you don't have to use the extension language. You don't need to learn JavaScript to use a browser, for example. But if you're a software developer, you do; otherwise, the browser won't do what you want it to do. If you want to use emacs in a way that isn't strictly out-of-the-box, you'll have to learn elisp in order to get your work done.
- We will only learn a subset of the basics of Lisp, just to get a feel of how to get something done.
- A nice property of lisp is because it's so simple, it can help expose some ideas about extension languages which would take us more time if we were doing this with JavaScript or the shell or something more complicated. It's a very simple language, in a sense.
- You have some **atomic values** vs. **conses**. Atomic values are individual values in our lisp program that you can't really take apart. You can look at them and print them out, but they have no further internal structure that we care about (at least for now). For our purposes, we'll even think of strings as being atomic values.
- In the elisp buffer (the scratch buffer), you can type an atomic value, then C-j. It will evaluate the expression and tell us the value we get. There are some

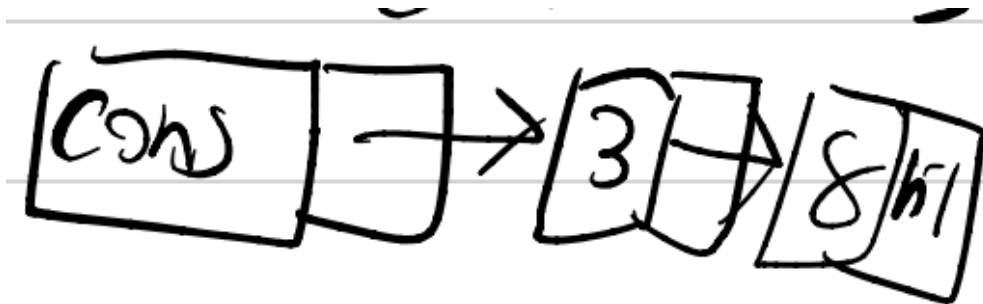
other atomic values that are special and worth mentioning.

- Numbers
 - Strings (not really atomic)
 - Symbols (e.g. `abc`). They look like identifiers and you can spell them like identifiers. If we want to have an expression that gives us that symbol that's a value, we can compute that symbol by writing the expression `"abc"`
- Symbols are in a sense new; we haven't seen them in C++ (they're not built into C++). An identifier in Lisp is an actual value. We can compute that symbol by writing
 - The simplest forms of Lisp are **self-evaluating** (e.g. numbers, strings, etc.). Other forms of Lisp are non self-evaluating, which include identifiers/symbols. What self-evaluating means: if you ask Lisp to evaluate a number, it will just spit back that number, as the number evaluates to itself. Non-self-evaluating includes identifiers. Symbols have values. If you ask Emacs the value of `abc`, you get an error because you haven't given it a value. It's a symbol that's unbound. If you ask it to give you the symbol itself, you need to surround it in quotes to indicate "don't evaluate the expression; just give me the expression as is." If we quote `abc`, it will evaluate to just the symbol `abc`.
 - `emacs -Q` means start up Emacs without your special configuration. By default, it will open up the scratch buffer which lets you type in Lisp code.
 - How do variables differ from symbols? Variables are typically named by symbols. e.g. we may say that `abc` has the value 3. What we really mean is that the variable whose name is `abc` has the value 3. We can give symbols of variables a value with `setq`. We can set variables like: `(setq abc -8)`
 - More on symbols: In Lisp, **symbols** are a fundamental data type used to represent identifiers, variable names, and function names. Unlike strings, symbols are uniquely stored, meaning two symbols with the same name refer to the same object. You can think of a symbol as "I'm just the name of a variable; I'm not its value." It's kind of an abstraction of a variable.
 - What we've done here is we've evaluated some forms in Emacs. Simplest form is self-evaluating. Just evaluates to itself.

- General syntax: `(setq symbol value)` which means "associate the symbol with the value of the expression `value`". It evaluates the expression and assigns the resulting value to the variable. It's like an assignment statement, except we don't have assignment statements in Lisp, we have `setq`. This command will return the value.
- Example: `(setq x 'abc)` Because we quoted `abc`, the expression is just `abc`. So the value of the variable `x` is just the symbol `abc`. We have two things going on here, names and values.
- The next idea about Lisp is the idea of the cons. A **cons** is the most basic data structure of lisp (there are plenty of other data structures in Lisp, but the cons is the one that people use the most often). It's pretty simple: it's just a pair of two other objects. You can put anything into a cons you like.
- How to create a cons: `(cons expr1 expr2)`. The way the cons is printed out is `(expr1 . expr2)`. All that Lisp does is create a new piece of storage, an object with two cells in it. The first cell has the value of the first expression and the second cell has the value of the second expression.
- How do we make this go further? By hooking up conses with each other. By convention, the way you write a list in lisp is by hooking up conses together.
- The end of the list is marked by a special value "**nil**"
- Self-evaluating forms include not just numbers, but also "nil." You can think of "nil" as being nullpointer or the empty list (Q: how does Lisp "nil" compare to C++ nullptr?)
 - In Lisp, nil is much more than just "null": it has multiple meanings depending on context, such as the empty list `()`, boolean false `(false)`, or a general "null" or "nothing" value
 - In C++, nullptr represents "no valid pointer" but nothing else. It is NOT equivalent to false or 0 in a boolean context. Prevents accidental integer-pointer confusion (nullptr is not 0)
 - So C++ nullptr is strictly a null pointer, while Lisp nil is much more flexible.
- How to build a linked list A, B, C: `(cons 'A (cons 'B (cons 'C nil)))`

```
(cons 'A (cons 'B (cons 'C nil)))  
(A B C)
```

- Writing (A B C) is equivalent to (A, (B, (C, nil))). They're equivalent expressions. The lisp interpreter likes to be concise so it will print out the (A B C) version (Q: compare and contrast these three notations). But note that typing either of those commands into the scratch buffer would fail. You'd need to write `'(A B C)` or `(list 'A 'B 'C)`
- This is how it might look like:



- That's how we build lists. How do we take them apart? By using car and cdr.
- `(car expr)` gives the first item of the list expr
- `(cdr expr)` gives the remainder of the list
- This is where the names came from: The IBM 79,4 had an address register and data register. How professor remembers it: he was driving down Sunset Blvd behind a brown Mercedes and the license plate on the back of the Mercedes said cdr because it was on the tail end of the car. The cdr function gives you the tail of the list. You can think of car and cdr as being head and tail.
- A cons is a pair of two objects. They can be any two objects. Notice that not every cons represents a list. (3 . -8) is not a list. A **list** is either the empty list (list with no items) or a cons whose tail is another list. That's the definition of a proper list in lisp.
- In the example above, A, B, and C are symbols. How do you evaluate them? Symbols are just blobs of data. The value of a symbol is the symbol. There is a

function that may be useful for what you want to do, but it requires one more idea:

- In lisp, every lisp program is represented as a list. Programs are always easily representable as data. A program is a piece of data. In the below example, "program" is a list containing cons, 3, and -8. Below, program is a piece of data. The data looks like a list with three items: cons, the number 3, and the number -8. We haven't actually called cons, it's just a piece of data that looks like a program.
- lisp has a special function called **eval** which will evaluate the program. eval tells the Lisp interpreter to take the piece of data, compile it into machine code, and then run the code. It's just an ordinary function in Lisp which will evaluate the program. So the distinction between the language and program, which is very sharp in a language like C++, is not so sharp in Lisp (Q: how is there a distinction in C++?). You can always create a program on the fly and execute it. It's very cheap and doesn't cost much, and people do it all the time. This notion of programs and data was first introduced in Lisp but it's so powerful and so popular that we see it in lots of other systems as well. Such as the shell. The shell has the eval operation as well. It's normal in many scripting languages, not just the shell or elisp. In Lisp, eval is to some extent more convenient though than it is in the shell, JavaScript, etc. because programs already look like data anyway and you don't have to worry about parsing strings and that sort of thing.

```
(setq ls (cons 'A (cons 'B (cons 'C nil))))  
(A B C)  
(car ls)  
A  
(cdr ls)  
(B C)  
(setq program (cons 'cons (cons 3 (cons -8 nil))))  
(cons 3 -8)  
(eval program)  
(3 . -8)  
(cons '"cons 8)  
('"cons . 8)
```



```
(eval (cons 'cons (cons 5 (cons -8 nil))))  
(5 . -8)  
(defun ma (a b c) (+ (* a b) c))  
ma  
(ma 9 3 -26)  
1
```

```
program='echo hello' # variable called program  
eval "$program goodbye" # built-in shell command eval  
# takes the string 'echo hello' and executes it  
hello goodbye # output. program was compiled on the fly and run
```

- How is the distinction between programs and data sharp in C++ but not in Lisp?
 - C++ has a strict distinction between code (compiled, immutable machine instructions) and data (manipulated at runtime)
 - C++ is a compiled language: The compiler translates source code into machine instructions.
At runtime, the program executes, but cannot modify its own structure.
 - Lisp blurs the line because Lisp code itself is just a type of data (lists), which can be constructed, modified, and executed at runtime.
 - Lisp treats code as data because Lisp programs are written in lists, which Lisp can manipulate just like any other data structure.

Quoting

- An important principle in shell, lisp, and other languages: you have to follow quoting. Quoting is a big deal, even though it may not look like it. Why did we put an apostrophe in front of the second cons in the above example and not the first? If we put it in front of the first and executed ('cons 3 -8), we'd get an error (you're trying to call a function, but "cons" is a symbol, not a function). You can only call a function, not a symbol. Quoting cons means we want the symbol cons; we don't want to know what its value is. For that reason, we can't put an apostrophe in front of that cons.

- When we want the symbol to be evaluated as a symbol, we have to quote. If we don't want that, we have to not quote it.
- `(eval 'cons 5 -8)` wouldn't work because we're giving eval 3 arguments, which is too many. What would work is: `(eval '(cons 5 -8))`. The `'` in front of `(cons 5 -8)` tells Lisp to not evaluate anything in there before giving it as an argument to eval.
 - How come `(eval '(cons 5 -8))` works, but `(eval (cons 5 -8))` doesn't? `(eval '(cons 5 -8))` works because `'(cons 5 -8)` is a quoted list, so eval sees `(cons 5 -8)` as an expression and evaluates it correctly. `(eval (cons 5 -8))` fails because `(cons 5 -8)` is executed first, producing `(5 . -8)`, and then eval tries to evaluate `(5 . -8)`, which is not a valid Lisp expression.
- Another way to make it work would be `(eval (cons 'cons (cons 5 (cons -8 nil))))` but no one ever writes that. It's totally impractical. Most of these examples are in fact not practical. The way you'd use them is if you're trying to build up a data structure inside emacs to build the list exactly how you like it, and for that quote is very helpful.
- Q: What about `(eval 'cons)`? Or `(eval cons)`?
 - Neither of these commands will work correctly because of how eval and symbol evaluation work in Lisp.
 - eval evaluates an expression. It expects a valid Lisp expression (a form that can be evaluated). It does NOT directly call functions—instead, it expects a full expression like `(cons 1 2)`, not just `cons`.
 - With `(eval 'cons)`, `'cons` is a quoted symbol, meaning it evaluates to itself. So, eval sees just `cons`, which is a function name but not a full expression
 - With `(eval cons)`, eval expects a full form like `(cons 1 2)`, but `cons` alone is just a function name.
- `(eval (cons 'cons (cons 5 (cons -8 5))))` would give an error because the provided argument isn't a valid program. You're trying to get it to evaluate something that you said would be a valid program, but it's not a list! Programs in Lisp have to be valid lists. Q: is it not a valid list because it doesn't have nil? Yes. To make it work, you'd either have to do `(eval (cons 'cons (cons 5 (cons`

-8 nil)))) which evaluates to (5 . -8), (cons 'cons (list 5 -8)) which evaluates to (cons 5 -8), or (eval (cons 'cons (list 5 -8))) which evaluates to (5 . -8)

- eval of the empty list isn't allowed either. Q: why? (eval nil) would fail because fails because Lisp expects eval to receive a valid expression, but the empty list is not a valid expression. (eval nil) may not actually cause an error though because the empty list (nil) evaluates to itself in most Lisp implementations, so when eval sees nil, it treats it as a value, not a function call. Since nil is already a valid value, Lisp just returns nil.
- Q: what happens if we try to get Lisp to evaluate an empty list? Error.
- This is the glue. How do we build things out of lists? With function calls. cons, car, and cdr are all function calls. The way function calls work is pretty simple: you evaluate all the arguments then go call the function. Not everything in Lisp is a function call (e.g. the self-evaluating numbers and exceptional things called **special forms**, which kind of look like function calls because they have parens around them, but they're not function calls). Q: what are special forms? It's a fairly small set. Usually when you see parens it's going to be a function call, but the special forms are exceptional.
- e.g. `setq` cannot be a function because if it were a function and the symbol didn't have a value, we'd get an error since we'd try to evaluate the symbol but the symbol wouldn't have a value. `setq` is special because it doesn't evaluate its argument the same way an ordinary function call would.
- More on special forms: Special forms in Lisp are fundamental expressions that do not follow normal evaluation rules. Unlike regular functions, they control evaluation of their arguments, allowing things like conditional execution, defining variables, and looping.
- What makes special forms special:
 - They do NOT evaluate all their arguments before execution.
 - They are built-in to the language and cannot be implemented as normal functions.
 - They enable core Lisp features like conditionals, variable bindings, and macros.
- Another special form is the command to define a function.

- You can define a function by doing: (defun f (a b c) ...code resulting in a b c...). `defun` is not a function (q: why isn't a function? Because it doesn't evaluate its argument) but rather a special form that will let you define a function.
- Q: how to define a function that adds C to the product of A and B. Try on your own first.

```
(defun addTwo(a b c) (+ c (* a b)))
addTwo
(addTwo 1 2 3)
5
```

- Emacs started out as a text editor. One of the ways it's still a text editor is, at the lowest level in the C code, it deals with people typing characters coming in (keystrokes). In that C code, there's a dispatching mechanism. When you type a keystroke, e.g. q, emacs figures out which emacs, lisp, or C function to call, then it goes and calls it. You can find out which function is invoked by typing `C-h k` followed by the keystrokes. e.g. `C-h k C-x h` which tells you that it's the `mark-whole-buffer` command. If you click on that it will give you the source code. Another way to invoke `C-x h` is `M-x mark-whole-buffer`.
- `C-h k` lets you describe a keystroke or mouse click. What would `C-h k x` do? It will tell you that if you just type in x in Emacs in this particular window, it's going to invoke a function written in C code. You can look at that C code if you're interested in looking at the lowest level of emacs. Prof hangs out at this level the most, but he doesn't recommend it if you're just getting started. (We did this example in the previous lecture).
- Let's look at a function that's not written in C. Try `C-h k C-x h`. It will tell you what happens when you type C-x h. It runs the `mark-whole-buffer` command and links to the source code. We see an emacs lisp function that's using the pattern we wrote earlier. `mark-whole-buffer` is a function that takes no arguments and does a whole bunch of stuff underneath. We can run that command by typing C-x h but there's another way to invoke it:
- There's a command `M-:` which lets you type whatever Lisp code you like. It's like going into the scratch buffer and typing some lisp code, except you can stay where you are without having to switch to scratch. e.g. you could

evaluate 2+2. Or you could evaluate mark-whole-buffer. It's a complicated way to do what C-x h would have done.

- The system we have here is one where we're looking at the source code of emacs while we're running the program.
- You can also do `M-x mark-whole-buffer`. What's the difference? Some emacs functions are designed to be hooked up to keystrokes or to be commands. The way you can tell whether a function is that way is if it's marked interactive.
- If a function is marked **interactive**: "I'm not only an ordinary emacs function; I can also be invoked from the top level by a keystroke." Not every function is like that. So functions marked interactive can be invoked by a keystroke.
- You can cut and paste a function definition into your scratch buffer. From there, you can change the function definition. e.g. instead of going to the end and start and marking that way, you could go to the start and the end. After editing the function, type C-j. Now C-x h would work differently for you (Q: how does it work? Can a function definition you wrote in the scratch buffer override the corresponding function in the source code? Yes! In Emacs Lisp (Elisp), you can override built-in functions or previously defined functions just by redefining them in your scratch buffer and evaluating them. This works because function definitions are stored dynamically in memory and can be changed at runtime. If you define a function with the same name as an existing one, it replaces the old definition. All calls to that function will now use the new definition, even if it was originally from Emacs' source code. But if you restart Emacs, the function reverts to its original definition unless you save your changes in your ~/.emacs or init.el file.) If you really liked it, you can put it into a file and compile it, maybe bind it to something else.
- The point is that you can change how emacs works fairly conveniently. You don't have to recompile all the source code or send in a bug report to the maintainer. You can just change it to be the way you like. Pretty much any good IDE will have that property. You should know how to do that with any good IDE because if you don't, you're just working with it like how everyone else is working with it, and you won't have a leg up on the competition.
- There's a function called `(global-set-key)` which lets you define custom emacs commands. e.g. here we can reprogram Emacs so that every time we type C-

`cq`, instead of doing what Emacs would usually do, it will call the `occur` function. To learn more about the `occur` function, do `C-h f`. The `occur` function looks for instances of regular expressions in the buffer. Now if you type `C-cq`, it will ask you "what regular expression would you like to match?" If you type "mini" it will show you all of the lines in the current buffer that have the string "mini" in them. The reason that worked is that you bound `C-cq` to be the `occur` function rather than what it normally is. If there's no existing command that does what you want, you can write your own command in the scratch buffer (or you can put it into an elisp file and then load the file) and then do the `global-set-key`.

```
(global-set-key "\C-cq" 'occur)
occur
;; What does the \ mean before C?
;; The \ in \C is an escape sequence in elisp strings that
;; you don't find in C.
;; in the ordinary elisp printer, \C prints as ^C. It's not
;; two characters ^ followed by C, it's a single character
;; whose binary value is just 3.
```

`occur` is an interactive byte-compiled Lisp function in 'replace.el'.

It is bound to `M-s o`.

(`occur` REGEXP &optional NLINES REGION)

Show all lines in the current buffer containing a match for REGEXP. If a match spreads across multiple lines, all those lines are shown.

Each match is extended to include complete lines. Only non-overlapping matches are considered. (Note that extending matches to complete lines could cause some of the matches to overlap; if so, they will not be shown as separate matches.)

Each line is displayed with NLINES lines before and after, or -NLINES

before if NLINES is negative.

NLINES defaults to 'list-matching-lines-default-context-lines'.

Interactively it is the prefix arg.

Optional arg REGION, if non-nil, mean restrict search to the specified region. Otherwise search the entire buffer.

REGION must be a list of (START . END) positions as returned by 'region-bounds'.

The lines are shown in a buffer named '*Occur*'.

That buffer can serve as a menu for finding any of the matches for REGEXP in the current buffer.

C-h m in that buffer will explain how.

- Emacs thus has a configuration capability that goes well beyond what we normally see.
- Emacs has a configuration system. You can look at `my.emacs`. It shows you the special things we want in our particular emacs. It's like setting a bunch of environment variables.
- Some of the stuff we want may be too fancy to be an environment variable. E.g. when we look at a file, if the file names are a certain kind, you want to do something special. You do that by writing a function. A lambda expression is an unnamed function that does what you want. Writing lambda functions gives us a lot more capability than just setting a bunch of configuration variables. (Q: is this similar to .profile in the shell? Yes.)

Arithmetic

- In Emacs, every operation that does anything is a function. So functions always use the prefix notation. * is an identifier, it's not a special symbol or anything like that. It just happens to be a built-in function that does multiplication. It takes any number of arguments. It also takes no arguments (if you multiply no numbers together, you get the identity element)
- If you multiply no numbers together, you get the identity element (1 for multiplication)

- Division takes the first number and then divides it by all of the other numbers. Special case: if you divide a single number, you get the multiplicative inverse
- The arithmetic operations are just functions. Some functions can take any number of arguments

```
(*)
1
(/ 3.5)
0.2857142857142857
(/)
error
(/ 1 0)
error
```

- `(/)` is invalid (q: why? Because there's no identity for division — division without arguments makes no sense). When something invalid happens, emacs will open a backtrace buffer in the debugger mode. It will give you a backtrace of all the functions you called to get to the bad one. There are all sorts of commands you can type in there; there's like a 30-page chapter in the emacs manual saying what you can do there. But Prof isn't an emacs debugger expert, he prefers to write code that doesn't need to be debugged.
- `C-J` is the command to stop debugging. To prof it is the most important part of the emacs debugger.

Python

- Prof feels weird teaching Python because most students are familiar with it. To some extent, he wants to teach the motivation and how to learn more stuff about it, without really going into a lot of the details. But it's impossible to talk about a software technology without actually having some details.
- Here's a line of input Prof got by scraping a website. You can think of it as being a stock trade where you trade 100 shares of some Alphabet stock (with the stock symbol GOOG) with a price of \$205.35 per share.

- `line = 'GOOG,100,205.35'` represents a stock trade. We can do something that is a bit like the eval function of Lisp in order to process this line. We can say “give me a list of types”: `types=[str,int,float]`. Think of it as being 3 columns in a table, and the line is a row in the table. 1st column is type string, second column is of type int, 3rd column is of type float. We can ask for the fields of the line in a form that we prefer, not just as a string of bytes as we have here, but as a string GOOG, an int 100, and a float 205.35.
- Give me the fields of this line in a form I prefer: `fields=[ty(val) for ty,val in zip(types, line.split(','))]`. Break down of what this does:
 - `zip(types, line.split())` creates pairs of (type, value), i.e.

```
[
  (str, 'GOOG'),
  (int, '100'),
  (float, '205.35')
]
```

- Then `fields = [ty(val) for ty, val in zip(types, line.split(','))]` does list comprehension where it loops over each (ty, val) pair, converts val to type ty, and stores the result in the fields list.
- This is a common trick for parsing structured data in Python!
- This would result in something like `str('GOOG'), int('100'), float('205.35')`.
- `zip` lets you interleave two values of two lists. We have two 3-item lists (one of the types and one of the line split into an array).
- So we have a list of tuples and we iterate through the list. Each time we iterate, we take this pattern.
- If you pass a string to the str type, it's just a constructor which gives you the string. It will call `int(100)` which just returns the number 100. Also calls `float` on 205.35 to give us the floating point value 205.35. The list comprehension construct says: take those three values you got by calling `ty(val)` three times and give me the list of what you got.

- Why didn't we just say `line` instead of `line.split`? Because if we just did `line`, we'd be iterating over the individual characters. We didn't want to think at that low level. We could write code that walks through that character by character, but that's what `split` is for. `split` is really good at doing what we want, which is splitting the line up into words separated by commas. We wanted to think abstractly rather than at the lowest level.
- This kind of programming is not uncommon in python. You're just hooking stuff together in kind of an ad hoc way (it's specialized, only works on this particular stock code, etc.) but if you try to do this sort of thing in C++, you'd have to write fairly specialized code. Whereas here you're just putting together stuff that's already there.
- So in some sense, you could think of python as an extension language in which the primitives are basic data structures like strings, lists, etc. and you're extending them to do what you want, but in another sense, Python is really not an extension language. It's intended originally to be a valid programming language in its own right. It had motivations for being the way it is. Even though people write code like this fairly commonly now, they didn't when python started, so it might be helpful to talk at least briefly about where it started before we ended up writing code that looks like this.

History of Python

- Python started out as a rebellion against **BASIC**. BASIC started out as a rebellion against the very first successful programming language, **Fortran** (which is still used for scientific computing). Fortran is to scientific computing as C is to computing with pointers and characters (C is a low level language, if you do anything at all wrong it crashes, etc.). The rebellion against Fortran came about because two professors at Dartmouth in roughly 1963 wanted to teach people how to program, but Fortran was too hard and if you do anything wrong, it crashes. They wanted something friendly. BASIC is designed as an instructional language (as its name would suggest). It was the first programming language prof learned. Over time, BASIC grew. Now there's VBasic (Visual Basic), for example. BASIC got too big. Also the problem with BASIC is because it was designed in the 1960s, it was full of 1960s constructs like "goto"s. So it ended up being bad for instruction. Then in the 1980s, a

group at Amsterdam wanted to come up with a better language for teaching, which they called **abc** to reflect that it's even simpler than BASIC.

- Ideas of abc:
 - We spent too much time in CS 31 learning about how to indent our code. That's a waste of time. Just have the system do it for you automatically! In abc, indentation is either automatic (if you're using an IDE, which they supplied for their students) or required (if you're using a random text editor like emacs, you have to indent so that the code is easier to read and grade). That's the advantage of abc over BASIC.
 - We spend too much time on quicksort and heapsort and linked lists. What a waste of time! When are you ever going to write heapsort code? Instead of wasting students' time, just have the system do sorting and lists automatically. The idea is that you have basic data structures and algorithms built into the system. Instead of learning what sorting and hash tables are, you'll learn, once you have sorting, what do you do with it? Instead of learning how to build a hash table, once you have a hash table, how can you use hash tables to build more interesting things? To some extent, what's going on here is that we're trying to program at a higher level of abstraction and avoid the low-level details. They tried this. They built a system called abc and shipped it out as floppy disks to all the high schools in the Netherlands. It flopped because the students said the jobs are in BASIC and no one will hire them if they know abc.
- Out of the wreckage of abc arose python! Python wasn't intended as an instructional language, it was intended to build out of abc (in the same way BASIC arose out of Fortran — you start out with a language that's easy to teach and easy to learn, but we make it practical and able to do a lot of the things that normally you'd waste a lot of time writing a C program to do). That's the underlying motivation. When you see some quirks in python that are hard to understand, you have to remember where it came from and use those origins as inspiration.
- Python values are objects. Every value in Python is an object. In that sense, Python is different from C++. In C++, 37 is a value but not an object; an object has to have an address.

- Q: does Python not have a distinction between primitives and objects? How does this compare with JavaScript? In JavaScript, everything is either a primitive or an object. In Python, all values are objects—there is no separate category of primitives like in Java or JavaScript.
- Every Python object has 3 things: identity, type, and value.
- The **identity** of an object roughly corresponds to what you'd call an address in C++. Every object has a unique identity, and no other object has that same identity. There's a built-in function in Python `id` : `id(27)` gives 8917232. You can think of that long integer as being kind of like a pointer. It's not really a pointer, but it kind of acts like a pointer, as it acts as a unique id for that particular object. If you assign the number 49 to a and ask for the identity of a, and then later ask for the identity of a, you'll get the same identity. The object always has the same identity
- Q: but what if you reset a to some other number? The identity changed. Why is that?
 - In Python, variable names (a, b, etc.) are just labels (references) to objects.
 - When you reassign a = 49 to another number, it points to a new object, which has a new identity (id).
 - The old number object still exists (if referenced elsewhere), but a no longer refers to it.
- `type()` function gives the **type** of the object. `type(a)` is `int`. Python is object-oriented, so we call it class, but it's really a type.
- Q: are types classes in Python? Yes.
 - In Python, types are implemented as classes.
 - Every type (e.g., `int`, `str`, `list`, `dict`) is actually a class.
 - Objects are instances of these classes.
- The third thing an object has is the **value**, in this case it's 27.
- `id` and `type` don't change; they're immutable. If you ask an object its identity now and later, the answer will be the same. Same for `type`; the type of an

object can't change.

- But the value *can* change. But only if the object itself is mutable. Not every object in python is mutable. If you do any serious programming in python, you should always know when you're dealing with an object that's mutable or not. They have quite different properties. So type and id are immutable, but value can change for mutable objects.
- e.g. [3,9,12] is a list. Lists are mutable. (3,9,12) is a tuple. Tuples are immutable. They can't be changed.
- This issue of mutability turns into a big deal. Whenever you're using a scripting language or an extension language, you should really know if you're dealing with objects that can change (Q: why is this important particularly for scripting/extension languages?)
 - Scripting languages (like Python, JavaScript, and Lua) are dynamic, meaning objects are often modified at runtime.
 - Mutability affects behavior when passing objects to functions, storing them in variables, and modifying them inside loops or collections.
 - Unexpected mutations can cause bugs if programmers assume that objects remain unchanged.
- In JavaScript, for example, JS started out with everything being mutable. Then, over the years, developers found out that's a lot of trouble (Q: why?). So the trend in modern JS libraries is to make objects less mutable. Same thing goes in Python.
 - Over time, developers realized that excessive mutability led to bugs, security issues, and harder-to-maintain code. As a result, JavaScript introduced features like `const`, `Object.freeze()`, and immutable data structures to help control mutability.
 - Mutability means that objects, variables, or functions can be changed after creation. When everything is mutable, unexpected modifications can cause hard-to-debug issues. JavaScript initially had no built-in mechanisms for preventing unwanted modifications.
- Q: What are the pros and cons of mutability?

- Pros:
 - More Efficient (Avoids Creating New Objects): Mutating an object modifies it in place, avoiding the overhead of creating new objects.
 - More Flexible for Real-Time Modifications: In data structures like databases or game states, mutability allows dynamic updates. Useful for cases where data is updated frequently.
 - Useful for caching and memoization: Mutability allows storing intermediate results, improving performance.
 - Allows for more intuitive OOP since in OOP, objects often need to maintain state.
- Cons:
 - Can Lead to Unintended Side Effects: If an object is shared across different parts of a program, modifying it in one place can unexpectedly affect other parts.
 - Harder to debug: Debugging becomes difficult when objects change unpredictably.
 - Makes code harder to reason about: If a function modifies an object, other parts of the program may unknowingly depend on it.
 - Causes concurrency issues: In multi-threaded or async programming, shared mutable state can lead to race conditions.
- Q: Is mutability the only difference between lists and tuples? No. It is the biggest difference, but there are also other differences, particularly performance, memory usage, and allowed operations.
 - Tuples are faster than lists because they have a fixed size.
 - Lists require more overhead due to dynamic resizing.
 - Since tuples are fixed in size, they use less memory than lists.
- The parentheses in (3,9,12) are not strictly needed. You can just write the tuple as 3,9,12 but most of the time people put parentheses around it to underscore the fact that it's a tuple as opposed to a list.

- With JS, you're working in a large program and have a large chunk of JS that you're copying off into the client/browser and it starts messing with objects. Some other part of the program wants to use those objects and gets annoyed because you're stepping on them. That can turn into a huge issue. Sometimes you want mutability, but you really want to know when the object is mutable and when it isn't (Q: when might you want mutability?)
- If you want to look at the properties of an object, objects have attributes and methods. If you have an object, you can assign or look up attributes using the dot operator (`o.a` where `a` is the attribute and `o` is the expression that yields the object). You can also invoke a method associated with an object using the dot operator. This idea works similarly to C++, except everything is done dynamically. If an object doesn't have an attribute you want it to have, you can just assign it (don't have to declare everything ahead of time like with C++).
- You can also inspect identity and value. There's a built-in expression in python called `is` that compares identities. `a==b` compares values. In C++, it's the distinction between `a==b` (comparing addresses of objects) vs. `*a==*b` (comparing the addresses of objects vs. comparing their contents)
- More on `is`:
 - `is` checks object identity (whether two variables point to the same object in memory).
 - `==` checks value equality (whether two objects have the same contents, even if they are different objects in memory).
 - `is` compares memory addresses (object identity). It returns True only if both variables reference the same object.
 - `==` compares the contents (values) of two objects, regardless of whether they are stored at the same memory location.
 - Sometimes, `is` and `==` return the same result if Python reuses the same object in memory. This can happen with small integers and strings that Python caches in memory.
- 💡 Much of the strength of python comes from its abc heritage in the sense that the built-in types and operators of python are very well thought out and

designed for ease of use and power.

- Python has a lot of built-in types.
 1. The simplest type is the **None** type for the value none. Can think of it as the null pointer.
 - a. Python's None is an actual object, whereas nullptr is a literal null pointer. None is an object of type NoneType.
 2. The second class of types are the numbers (int, float, complex).
 3. Next class of types are the sequences. There are a whole bunch of sequence types in python, including strings, tuples, lists, and more. Each one of these types supports all of the following operations:

Operations on Sequences

- Built-in operations on sequences in python:
 - `s[i]` gets the ith element of the sequence. This is the most basic sequence operation and all sequences must support it. Looks like a subscript notation. The model here is that your sequence of length n starts with item 0 and goes up to item n-1. If the value i is in the range 0 through n-1, you just get the appropriate slot. But python has an extra wrinkle here: if you prefer, you can also number the sequence using negative indices, where the last entry is the -1st entry and the first entry is the -nth entry. You can provide negative indices -1 (last element) through -n (first element). The rule here is $-\text{len}(s) \leq i < \text{len}(s)$. If you violate this rule, it's a runtime error. Python will yell at you and throw an exception.
 - `s[i:j]` selects a subsequence out of the original sequence. You start at element i and go up to but not including element j. The length of the subsequence is j-i (assuming i and j are non-negative)
 - `s[i:]` means go to the end of the list starting from i. Equivalent to s starting from i up to the length of s: `s[i:len(s)]` (Q: does this support negative indices?)
 - `s[:j]` means start at the start of the list and go up to but not including j: `s[0:j]`. i and j can be negative, by the way.

- `s[:]` means `s[0:len(s)]`; gives everything in the list. Can be used to copy the list.
- Length of any sequence: `len(s)`
- Min and max value of any sequence: `min(s)` , `max(s)`
- `list(s)` gives a copy of the sequence as a list (e.g. if you pass in a tuple, you'll get a list out of it)

```
s = "abcdef"
```

```
print(s[-3:]) # ✓ "def" (Starts from the 3rd-last character)
print(s[:-3]) # ✓ "abc" (Up to but not including the 3rd-last character)
print(s[-4:-2]) # ✓ "cd" (From the 4th-last to the 2nd-last, exclusive)
```

- The above operations apply to any sequence. We also have mutable sequences.
 - `s[i] = v` means replace the *i*th entry with whatever *v*'s value is
 - `s[i:j] = s1` means replace the entries *i* through *j* with the entries of *s1*. (Q: what if the lengths are incompatible?) This operation can change the length of the sequence. Entries *i:j* get deleted and replaced with a copy of what was in *s1*. (q: what happens to *s1*? Does it change if *s* changes?)
 - If *s1* has a different length than *j - i*, Python adjusts the sequence size dynamically.
The original sequence (*s*) is modified, replacing the slice *s[i:j]* with *s1*.
 - `del s[i]` is equivalent to saying `s[i:i+1]=[]`
- Suppose we have:

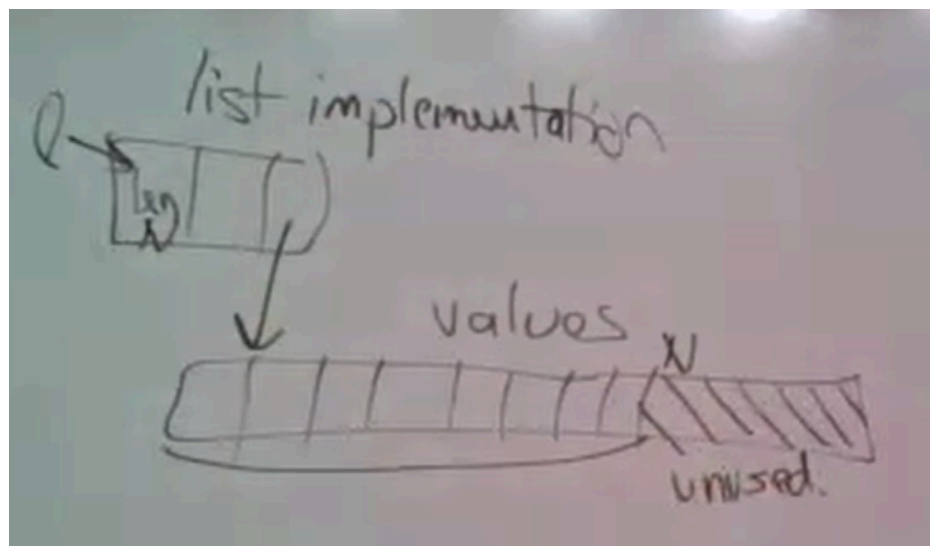
```
ls=[10,20,30,40]
lt=ls
lu=ls[:]
```

- What's the difference between `lt` and `lu` ? In the above example, `lt` is just a reference to `ls` ; changing `lt` will change `ls` . But `lu` is a copy of `ls` .

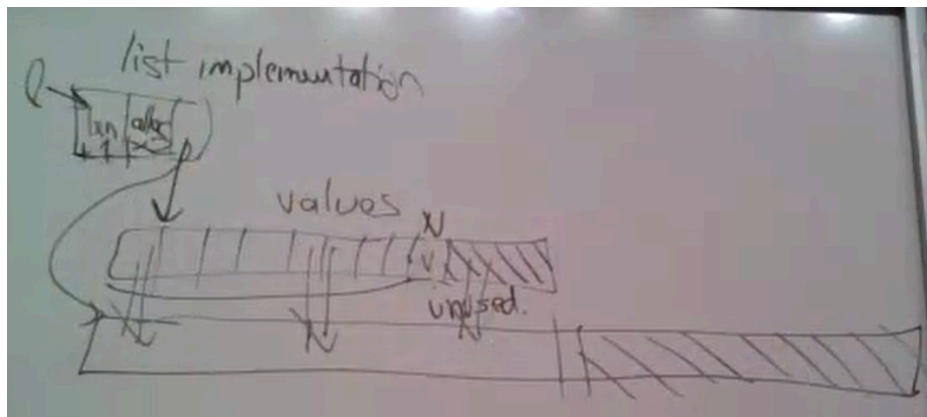
- Note that `s[i:j]`, `s[i:]`, `s[:j]`, `s[:]` give you a copy of what's in the original sequence. Whereas `l=l` just assigns one python variable to the other; it doesn't make a copy of everything in the variable. It's object-oriented and you're just passing addresses. Ordinarily, python likes to do that because it's faster. But if you really want to make a copy of a top level of a sequence, use the `[:]` operator to get a fresh object that you can then manipulate without affecting the object you subscribed off of.

Append

- Lists are the most commonly used sequences in python.
 - `s.append(v)` grows a list by length 1 by evaluating `v` and tacking it onto the end of the list. This operation is fast, $O(1)$ amortized. Amortized means if you call the function N times, the cost of all the N times taken together will be $O(N)$; it doesn't mean it's $O(1)$ each time. It could be that there's a glitch and one is more expensive. But if you call it over and over again on the same list though, on average it will be $O(1)$. Why is this the cost of appending to a list? To know that, you need to have some intuition of how lists are implemented in python. A list is an object, but that object contains information about the list, like its length, and maybe some other info, and then a pointer to the actual value. If the length is N , then the first N slots will be occupied, and the remaining slots in memory will be unused.



- Ordinarily, you take the value of v , plop it into the first empty slot, increment the length by 1, and you're done in $O(1)$ time since it's just a store, a load, and a store. But the catch is when you run off the end. `alloc` tells you the total number of spaces allocated in the array, as opposed to the number of spaces currently being used. The problem is when `N = alloc` and we call append. So append has to ask the OS for more memory, e.g. twice as much, copy all the values from the old storage to the new one, update the pointer to point from the old place to the new one, and appends the new element. So `alloc` grows by a factor of 2 while `N` only grows by 1. Can you prove that the amortized cost of this approach is $O(1)$? If we do this over and over again, where each time we grow the limit, grow the amount of storage by a factor of 2, how much total work do we do? Suppose we grow the list to a billion entries. How much did it cost us to grow the list to be that long? The answer is that it didn't cost us all that much; it's $O(1)$ per call to append. Suppose we have 2^{30} entries (roughly a billion). Of those entries, half were done in $O(1)$ time since they just had to be plopped there. The other half cost more because they had to be copied over. So the total cost of building the array will be, per entry, $1 \cdot 0.5 + 2 \cdot 0.25 + 3 \cdot 0.125 + \dots$ It's a geometric series. You can use a simple recurrence relation to figure this out. It will add up to a finite number, meaning the overall time complexity is $O(1)$ amortized.



- **Doubling Strategy:**

Every time the list is full, its capacity doubles. Suppose you start with a capacity of 1. The sequence of capacities is:

$$1, 2, 4, 8, \dots, 2^k$$

where 2^k is roughly the number of elements n in the list at the end.

- **Cost of Copies When Resizing:**

- When the list grows from size 1 to 2, you copy 1 element.
- When it grows from 2 to 4, you copy 2 elements.
- When it grows from 4 to 8, you copy 4 elements.
- In general, when it grows from 2^i to 2^{i+1} , you copy 2^i elements.

- **Total Copying Cost:**

If you perform n appends, where $n \approx 2^k$, the total cost of all the copying operations is:

$$1 + 2 + 4 + \dots + 2^{k-1} = 2^k - 1 \approx n - 1$$

Thus, over n appends, the total extra cost for resizing is about n .

- **Total Cost of All Appends:**

Each append normally costs 1 unit of time, plus the occasional extra cost for copying. So the total cost $T(n)$ is roughly:

$$T(n) \approx n \text{ (normal appends)} + n \text{ (copying cost)} \approx 2n$$

Therefore, the **amortized cost per append** is:

$$\frac{T(n)}{n} \approx \frac{2n}{n} = 2,$$

which is $O(1)$.


3. The Geometric Series Argument

Your professor mentioned a geometric series like:

$$1 \times 0.5 + 2 \times 0.25 + 3 \times 0.125 + \dots$$

This series is a way of thinking about how often each "cost" is incurred per element:

- **Half the appends are "cheap":**
For roughly half the appends, no reallocation is needed; they cost 1 unit.
- **A quarter of the appends incur a cost of 2 (because they trigger a copy of a block of size 1):**
For about 25% of the appends, you pay an extra cost.
- **An eighth incur a cost of 3, and so on.**

Mathematically, the sum of this geometric series converges to a constant. The exact sum isn't as important as the fact that it **converges**, meaning that even when you add up all the extra costs, the extra work per append stays bounded by a constant factor .


Explanation with Formal Recurrence Relation

2. Setting Up the Recurrence Relation

Let $T(n)$ be the total cost of performing n appends, where n is a power of 2 (for simplicity). There are two kinds of operations:

1. **Regular Append (when there is space):** Costs a constant c per operation.
2. **Resizing:** When the list reaches capacity $n/2$ and we append the $n/2 + 1$ -th element, we must copy $n/2$ elements. We will consider the cost of copying as proportional to the number of elements copied (say cost = $k \cdot (n/2)$), but for our asymptotic analysis, we can think in terms of $O(n)$.

For our recurrence, we can simplify by assuming that a resize operation costs n units, and that appends that do not trigger a resize cost a constant (which we can incorporate into the overall cost without affecting the asymptotic behavior).

Because the list doubles its size every time it runs out of space, if we have built a list of n elements, the previous resize occurred when the list was of size $n/2$. At that time, copying cost $O(n/2)$ was incurred. Then the cost to build the first $n/2$ elements is $T(n/2)$, and then we have the extra cost of copying the $n/2$ elements when growing from $n/2$ to n . .

Thus, the recurrence can be written as:

$$T(n) = T\left(\frac{n}{2}\right) + n,$$

with the base case $T(1) = c$ (a constant).

3. Solving the Recurrence

We can expand the recurrence:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + n, \\ &= \left[T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n, \\ &= T\left(\frac{n}{4}\right) + \frac{n}{2} + n, \\ &= \left[T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + \frac{n}{2} + n, \\ &= T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n, \\ &\vdots \\ &= T(1) + n \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{n}\right). \end{aligned}$$

The series $1 + \frac{1}{2} + \frac{1}{4} + \cdots$ is a geometric series with ratio $\frac{1}{2}$ and it converges to 2.

4. Amortized Cost Per Append

Since $T(n)$ is the total cost for n append operations and $T(n) = O(n)$, the amortized cost per append is:

$$\frac{T(n)}{n} = O(1).$$

