# Admin input from Andrew

* Found that 3 weeks was too long, so **we will meet every 2 weeks from now on**. If a certain chapter is especially hard, we may take 3 weeks for that one.
* We will be skipping the summer holidays (August).
* Types of homework: Labs / homework / projects / exams
  + Labs are meant for **before** the week’s reading
  + Homework exercises. Carefully selected and not repetitive, each is important.
  + Projects: Larger assignments. 4 during the “semester”: First 2 individual, last 2 pairing up. **We should try to find a partner.**
  + Exams. We’ll be taking them!

# Discussions

## Setup for the course

### Tech

* Some people are using emacs. They like that it provides some benefits – the key mappings in emacs are also available in bash. Also that it’s the same they use for other languages.
* Other people had trouble trying to set up emacs.
* Andrew has explicitly decided to ignore all Bash, Emacs, Unix stuff and use exclusively DrRacket, in order not to dedicate intellectual power for additional setups.
* Several people agreed DrRacket is a great IDE.

### Content

* Can exercises be skipped? Not recommended. There is not much repetition, each exercise is meant to touch on something important.
* Someone has been watching the original lectures, 1986 (https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-001-structure-and-interpretation-of-computer-programs-spring-2005/video-lectures/). Finds them more interesting. Eg. there is a deeper discussion about **how** functions returning other functions enables cool things like creating an image.
* For some people the original MIT lectures were too dense, going too quick to deep technical discussions and getting too complex without teaching Lisp first.
* The Harvey lectures are more accessible, they take you by the hand. Also they actually take time to explain Lisp.
* Nowadays MIT and Berkeley both teach the class with Python. They got too many complaints about not using a language used in the real world. But originally the goal was focused on abstractions, not the language.
* **Reminder:** The Harvey lectures are the official ones for this study group.

## Tips and tricks

* Pain point: the lectures have bad quality. => Trick: opening the code in IDE separately, so you can see it while he presents.
* Trick for Lisp parentheses: you can use different types of brackets. If you use square brackets, the interpreter will consider them to be normal brackets.
* Simply scheme docs: <https://docs.racket-lang.org/manual@simply-scheme/index.html>
* Math videos: <https://www.youtube.com/channel/UCYO_jab_esuFRV4b17AJtAw>
  + Engine for precise programmatic animations: <https://github.com/3b1b/manim>

## Learning Lisp

* A couple of people noted that it seems to invite writing small helper functions, separating everything into small pieces of functionality. This is one aspect of abstraction called functional abstraction that we are trying to learn about.
* When you learn a new language, you have the tendency to write the way you’re used to, e.g. “you can recognise Python written by a Java programmer.” That doesn’t work in Lisp, for example because you don’t set flags.
* The heavy use of recursion is fascinating, but also scary.
* For learning scheme, Brian Harvey wrote the book “Simply scheme”
* Another nice book: The little schemer. Easy, light-hearted way to get around the Scheme concepts. Someone expressed Little schemer is helping them with recursion, which they always found difficult.

## Difficulties

* A couple of people had difficulty with exercise 1.6 (why “if” needs to be a special form and can’t just be replicated with a normal function “new-if”). They thought they understood the evaluation order but were confused applying it to this problem initially.

## Things we liked / found interesting

* The general philosophical and conceptual points, e.g. “A programming language as a medium for organising and expressing procedural thoughts”. Identifying on a conceptual level the fundamental components needed for a powerful programming language (primitives, means of combination, means of abstraction.)
* Understanding that having lambdas, i.e. being able to have functions as first order citizens essentially allows you to build everything else.
* Functions as black-box abstractions. Once you put it behind a name, you forget about the implementation.
* Provocative concept that “Computer science” is the wrong term, that we are not talking about a science.
* Concept that programs should be written for humans, not computers.

## Theoretical discussions about Lisp and other programming languages

**Note:** You can skip this section without missing anything about the study group’s progress. It was an interesting, nerdy discussion.

### Contributions about Lisp

* Impressed how compact the language is.
* Cool that it is self-reflecting. You can write about it in itself. If you try to do that in C or other languages, you are limited.
* Powerful feature: the first-class citizenship of lambda functions.
* Some influential people (uncle Bob) saying Lisp might be a language of the future. Original problem with it was it consumes a lot of memory, not so important now.
* The fact that it’s basically functional seems appropriate to where we’re heading to.
* Why is Lisp not more used? Opinion: Possibly due to the heavy use of recursion.
* It excels for domain-specific languages.
* The great thing with Lisp is it’s really simple. “List processing.” Other languages are very powerful but not that simple.
* Are special forms a feature of the language? In reality just a single feature of the language: macros. Differently to other languages like C, Lisp macros are full-blown Lisp procedures, with the full power of any other Lisp procedure
* It is a very pure language. Great for learning programming.
* You don’t worry about the hardware if you use Lisp. As mentioned in one footnote: Alan Perlis (paraphrasing Oscar Wilde): ‘Lisp programmers know the value of everything but the cost of nothing.'
* **Reminder: It’s not the purpose of this book / study group to learn Lisp. It’s about the concepts.**

### “Better or worse” discussions

* Some arguments that Lisp is “better” / “more powerful” than other languages. Counter-arguments: it’s not better, just a different tool, different uses.
* Felix, who has a lot of experience with functional programming felt Lisp was overhyped. Feels Erlang, or Elixir have a step more into reality, more powerful, allows you to do more real world stuff. Lisp is more academic.
* Others felt that Lisp can solve things that modern languages cannot. Someone agreed he doesn’t miss much in Lisp.
* In any case, surprised that a language from the 70s might be considered (in some ways) better than languages built in the 90s.
* “Languages of the 70s 80s were very small, low quantity of concepts that you can combine to produce more complexity.”

### C, Java

* The book says there is a flaw with the language C. Someone who works with it for a long time expressed interest in finding out what it was.
  + Someone else understood the flaw with C is about tail recursion. Big overhead by having to keep the stack, which can lead to stack overflow.
  + Another input: it’s about compiler optimisation in recursion. However, not all Lisp has compiler optimisation so it’s not an actual Lisp feature.
* Java didn’t have closures for 20 years!

### Javascript:

* Cool to see that a lot of Javascript comes from Lisp.
* Javascript was originally written in 9 days, and the original intent was to port Scheme! It has lots of things coming from Lisp, like closures, tail recursion, etc
* They then made the syntax look like C. Aditionally people try to warp the functional core of Javascript into something looking like Java. Classes, etc. But actually, if you take out all those attempts on top to make it look Object Oriented, it’s functional based.
* Lots of books coming out about functional programming with Javascript. Coming to the original intention.
* One thing that used to be very frustrating about Javascript was it didn’t have lexical scope.

Chat from Catriel (I don’t understand the second part):

*“It's not only functional; javascript has some ideas of Self, it's prototyped object programming. Where messages, resolve when a slot with a method definition is found in the hierarchy. Similar to dynamic scoping.*

*I think that a fair quantity of misunderstanding comes from there.”*

### 

### Domain-specific languages

* Some people are especially interested in domain-specific languages, eg. for AI. Something in which Lisp excels.
* Another input: big fan of Elixir, another language in which you can do domain-specific. Other languages also do that job.