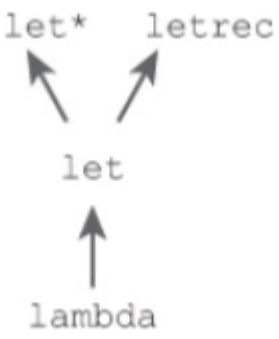
exam 2

Chapter 5: Functional Programming

- What is the distinction between a formal parameter and actual parameters? (Equivalent wording: between bound variables and arguments)
  - formal parameters (known as bound variables or simply parameters)
    - are used in the declaration and definition of a function
  - actual parameters or arguments
    - are passed to a function in an invocation of a function
- Why is adding to a list with cons more efficient than append?
  - `cons` - Constant O(1)
    - adds an element to the front of a list in constant time O(1)
    - it doesn't need to traverse the list to add an element
  - `append` - Linear O(n)
    - takes two lists and concatenates them for the second list to be added to the first `append` need to traverse the first list, making it a linear-time operation O(n) where n is the length of the first list
  - `reverse1` - Quadratic O(n^2)
- What is the difference-list technique?
  - an optimization where a list is represented as a function that takes another list and returns the original list concatenated with the argument
  - this avoids the costly traversal of lists when using `append`
  - the difference-list technique allows for efficient concatenation by deferring the construction of the list until necessary
  - **run time of the function is linear in the size of the list to be reversed**
- We won't get into the "shortening" call methods of Scheme shown in table 5.1, because Racket has commands like second, third, fourth, etc. But make sure you understand them, and the take, list-ref, and list-tail functions.
  - `take`
    - extracts a certain number of elements from the start of a list
    - return the first n elements of L
  - `list-ref`
    - returns the nth element of a list
    - returns the item at position pos
  - `list-tail`

        - returns the sublist starting from the nth element
        - returns the remainders of the list after the first pos elements
- Use of let, let*, and let-rec functions
    - `let` - bindings are added to the environment in parallel
        - introduces local bindings in parallel
        - in the list of lists immediately following `let`
        - and are only bound during the evaluation of the second S-expression
        - use of `let` does not violate the spirit of functional programming for two reasons
            - `let` creates bindings, not assignments
            - `let` is syntactic sugar used to improve the readability of a program
        - any `let` expression can be rewritten as an equivalent `lambda` expression
        - useful for naming intermediate results
    - `let*` - bindings are added to the environment in sequence
        - similar to `let` but allows sequential binding
        - where each binding can reference previous ones
        - we can produce sequential evaluation of the bindings by nesting `lets`
        - a `let*` expression is syntactic sugar for this idiom, in which bindings are evaluated in sequence
    - `letrec`
        - used for defining recursive functions within a `let` form
        - where the bindings are mutually recursive
        - use the `letrec` expression to make bindings visible while they are being created
    - both `let*` and `letrec` are syntactic sugar for `let`
    - both `let*` and `letrec` are syntactic sugar for `lambda` (through `let`)
    - reduce the preceding `letrec` expression for `length1` to a `let` expression
    - functions only know about what is passed to them, and what is in their local environment



    -

- Principles of recursion:
  - base caser
  - recursive steps
  - lists
  - numeric values
  - use `cons` rather than `append` to build lists as `cons` is more efficient
  - use `let` to name recomputed subexpressions avoiding repeated evaluation
  - nest local functions to encapsulate lofic and reduce code repetition
  - factor our constant parameters for clarity and reuse
  - difference lists technique is used to send the solution so far down the recursive chain improving efficiency unnecessary list constructions

  What hare gave
  - Solve for the base case. Solve for n in terms of solution for n-1.
  - For lists, the base case is an empty list and the recursive step is handled in the else clause. For numeric values, the base case is usually 0 or 1.
  - Use cons rather than append to build lists.
  - Use let to name recomputed subexpressions, to avoid re-evaluation
  - Nest local functions (information hiding)
  - Factor out constant parameters
  - Difference lists technique (send “solution so far” down the recursive chain)
  - Correctness first, simplification second
- Wlaschin’s principles of functional programming:
  - Composition everywhere
    - use functions as building blocks using the output of one function as input to the next design them so you can snap them together like legos
    - build complex behavior by composing simple functions
  - Strive for totality
    - your function should do something sensible for any valid input if your numeric function crashes if passed a zero either modify it to do something sensible (use maybe or either, perhaps) or document that the parameter must be nonzero its up to the caller to check that its valid
    - functions should be total, meaning they handle all possible inputs
  - Don’t repeat yourself
    - if different functions are doing almost but not quite the same thing factor out the common part and use sub functions or helper functions as needed
    - avoid repetition by creating reusable functions
  - Parameterize the things
    - generalize your functions by adding parameters a sorting function that can take in any comparison method is more useful than one locked in to using <

        - abstract over varying parts of code by passing them as parameters
    - The Hollywood Principle (don't call us, we'll call you)
        - provide callback functions so you can use partial application to customize a generic function
        - make your function reusable and wait for them to be invoked as needed
- Terms:
    - first-class entity
        - an entity that can be passed as an argument, returned from a function, and assigned to a variable (functions are first class in functional programming)
    - side effects
        - changes in state or observable interactions with the outside world that occur when a function is executed (pure functions avoid side effects)
    - lambda expression
        - an anonymous function that can be defined in-line (without a name)
    - S-expression
        - a symbolic expression, a notation used to represent nested list structures, especially in lisp like languages

---

Chapter 6: Bindings and Scopes

- Your text states "a closure can be though of as a pair of pointers." Where do they point?
    - closure
        - a function that remembers the lexical environment in which is was created
    - a closure can be thought of as a pair of pointers
        - one to a block of code - defining the function
        - one to an environment - in which function was created
    - a closure encapsulates data and operations and thus bears a resemblance to an object as used in OOP
    - the bindings in the environment are used to evaluate the expressions in the code
- Consider the languages: For each, does the language use static or dynamic scoping?
Static	bindings are	*fixed*	*before*	run-time. Example:	`int a;`
Dynamic	bindings are	*changeable*	*during*	run-time. Example:	`a = 1;`
    - C++
        - uses static scoping (lexical scoping)
        - scope of the variable is determined by the structure of the program code

    - Java
        - uses static scoping
        - variables scopes are determined at a compile time based on the program structure
    - Racket
        - uses static scoping
        - it determines the variable scope based on the lexical context
    - Python
        - static scoping
- What is an environment? Why is it important?
    - environment
        - is a mapping or collection of variable bindings and their values are available to a function or expression
    - importance
        - determines which variables and their values are available to a function or expression
        - crucial because it forms part of a closure which allows functions to retain access to the scope they were defined in even after the scope has exited
- Be able to find the lexical address of an expression in a nested function.
- Explain the difference between a bound variable and a free variable. If an expression has no free variables, what does that imply about its semantics?
    - Bound variable
        - declared within the function or scope in which it is used
    - free variable
        - used in a function but is not declared in that function
        - declared in an outer scope
- Advantages/disadvantages of static scoping; advantages/disadvantages of dynamic scoping. Which have most languages adopted? Why?
    - static scoping
        - advantages
            - scope determined by program structure
            - functions retain access to environment in which they were defined which allows for closures
        - disadvantages
            - can lead to less flexible code if you need yo dynamically change the environment during execution
    - dynamic scoping
        - advantages

        - more flexible as variables are looked up in the current call stack not in the lexical scope
    - disadvantages
        - harder to debug and reason about as a variables value can change based on the call stuck at runtime
- What is the FUNARG problem? Why does it arise in languages where functions are first-class entities?
    - FUNARG Problem
        - when functions are fist class entities (meaning functions can be passed as arguments or returned as values) - it relates to how variables in the functions environment are captured when the function is passed as an argument
    - Upward
        - occurs when a function is passed upward (returned has a value) to a higher level scope
    - Downward
        - occurs when a function is passed downward as an argument to another function
    - it comes up because first class functions can reference variables from the environment in which they were created which complicates managing variables lifetimes and scope
- How is a closure similar to an object? How is it different? (Note: Slides will probably not be adequate here… you might have to actually read the book….)
    - closure/object
    - similar
        - both a closure and an object bundle together data and behavior
        - in closure the data is the environment that holds the variables and the behavior is the function itself
    - differences
        - object encapsulates state and can have multiple methods
        - while a closure captures the environment for a single function
- What is the difference between deep binding, shallow binding, and ad hoc binding?
    - deep binding
        - the environment is bound to the function when the function is defined (lexical scoping)
    - shallow binding
        - the environment is bound to the function when the function is called (dynamic scoping)
    - ad hoc binding

        - functions environment is not clearly defined resulting in unpredictable or context dependent behavior
- Terms:
    - closures
        - a function along with its environment allowing the function to access non local variables even after the environment in which is was defined has exited
    - first-class closures
        - closures that can be passed as arguments returned as values and assigned to variables
    - static v. dynamic scoping
        - static scoping
            - determines the scope of variables based on where they are defined in the code
        - dynamic scoping
            - determines scope based on the runtime call stack
    - lexical addressing
        - a method of resolving variables references by looking up their location in the lexical scope
    - upward & downward FUNARG problem
        - challenges of passing functions as arguments or returning them as values involving capturing and managing their environment
    - denotation
        - the explicit meaning or reference of a function or variable
    - binding
        - associating a variable with a value of function
    - scope
        - the region of a program where a binding is valid
    - lexical scoping
        - another term for static scoping where the structure of the code determines variable scope
    - scope hole
        - a gap in the visibility of a variable often caused by variable shadowing or changes in scoping rules
    - ancestor blocks
        - outer blocks of code that enclose a nested function, containing the bindings that the nested function may reference

---

In class examples

- parametric polymorphism
  - different data types passed into a function

- lexical addressing
  0 1 2
  - (define (f1 a b c) 2
    0 1 2
    - (define (f2 d e f) 1
      0 1 2
      - (define (f3 g h a) 0
        - (+ d g a b)) <---- line z
  - (+ (1 0) (0 0) (0 2) (2 1))

---

closure and scope

- closure
  - a function and a reference to an environment
- (define (f x)
  - (define (g y) when this is called f of x has long since finished ---- so it needs to go in a closure (g y) \n ref to x --- we have a reference to x after long since its passed --- a way to implement lazy evaluation
    - (+ x y))
- call g)

---

ad hoc binding

- (define (f x)
  - (define (g y)
    - (+ x y))
- g)
- at some point we call f with some value at x
  - (f 2)

- deep binding we use the value when we created it
- at some later point
    - we cal (g 3)
    - shallow binding
    - what value does x have now?
    - racket uses deep
- shallow binding
- deep binding

---

first class items and entities

- can be passed into function has parameters
- can be returned from functions
- and can be assigned to a variable
- first order function
    - means we can pass a function
    - function returns a function
    - assign function to a variable

---

upward and downward funarg

- downward
    - which value gets passed down is the downward argument
- upward
    - we pass the function back up but we need this variable later

---

Chapter 7 Type systems

- Distinguish between:
    - type coercion v. type casting
        - type coercion
            - an implicit conversion that the language automatically performs to make types compatible in an operation. for example adding an integer to a float may cause the integer to be automatically coerced to a float

        - type casting
            - an explicit conversion where the programmer specifies that a value of one type should be treated as another type
                - ex. explicitly converting a float to an integer using a cast
    - parametric polymorphism
        - allows a function or data structure to be written in a generic way so it can handle values of any type
        - in racket and scheme it allows the creation of functions that operate any type without needing to rewrite them for each specific type
            - ex. a generic list function that can handle a list of integers floats or any other type
    - type inference
        - is the ability of the compiler or interpreter to automatically deduce the type of an expression based on context in which it is used
        - program does not need to explicitly specify the types
    - function overloading v. overriding.
        - function overloading
            - refers to defining multiple functions with the same name but different parameters- either in number or types
            - the correct function is chosen based on the type and number of args passed at the call site
        - function overriding
            - involves redefining a method in a subclass that exists in a parent class where the method signature remains the same
            - the subclass version will be invoked instead of the parent class version
- Advantages of
    - static typing over dynamic typing
        - static
            - advantage
                - early error detection
                    - type errors caught at compile time
                - performance
                    - statically type languages often perform better since the compiler has more information and can optimize code
                - better tooling
                    - enhanced support for IDEs with better autocompletion refactoring and error detection
            - disadvantage

        - less flexible
            - more rigid in how code must be written
            - all types must be declared explicitly or inferred
        - verbose
            - often requires more boilerplate code for type annotations
    - dynamic
        - advantage
            - flexibility
- Terms:
    - static typing
        - types are checked at compile time
        - variables must be explicitly declared or inferred with a type
    - dynamic typing
        - types are checked at runtime
        - the type of a variable is determined during execution and type errors only surface during program execution
    - parametric polymorphism
        - a form of polymorphism that allows code to be written generically so it can operate on values of any type
        - functions or data structures parametrized buy type
    - type inference
        - the ability of a compiler to deduce the types of expression automatically based on the context without requiring explicit type annotations
    - function overloading
        - defining multiple versions of a function with the same name but different parameter lists
        - type of number of args
    - function overriding
        - redefining a method in a subclass that exists in a parent class
        - the overridden method in the subclass replaces the parent version for that subclass
    - strongly typed
        - a language where types are enforced strictly and system does not allow implicit type conversions
        - ex. trying to add an integer to a string would result in an error
    - weakly typed

    - a language where types are not as strictly enforced allow implicit type conversions that can lead to unexpected results
    - ex. trying to add an integer to a string would result in an error
- sound v. unsound (safe v. unsafe) type systems
    - sound (safe)
        - type systems guarantee that there will be no type errors at runtime
        - all operations are type safe
    - unsound (unsafe)
        - type systems do not guarantee that runtime type errors will not occur
        - they allow potentially unsafe operations
- explicit v. implicit typing
    - explicit typing
        - the programmer must explicitly declare the type of variables
    - implicit typing
        - the language uses type inference to determine the type of variables automatically
- explicit v. implicit conversion
    - explicit conversion
        - the programmer manually converts a value from one type to another
        - casting a float to an integer
    - implicit conversion
        - the language automatically converts types as needed
        - promoting an integer to a float in a mixed type arithmetic operation
- monomorphic
    - functions or data structures that operate on a single specific type
    - the opposite of polymorphism
- type inference.
    - process by which a programming language automatically determines the types of expressions without explicit type annotations from the programmer
- lexical scoping
    - the scope of a variable is determined by its position in the code d
- dynamic scoping
    - the scope of a variable is determined by the runtime call stack where the function was called from
    - most modern languages use lexical scoping

Chapter 8. Currying & Higher-order functions

- What is currying? Why is it useful?
  - currying
    - transforms a function with multiple args into a chain of functions each taking one arg
    - instead of passing all the args at once you pass them one by one
  - useful because
    - it allows for partial application where some args can be fixed in advance creating specialized versions of functions
- How do higher-order functions expand possibilities in program development?
  - higher order function
    - takes one or more functions as args
    - returns a function as its result
  - these functions enable powerful pattern for abstracting and code reuse some well known higher order functions include
    - map applies a function to each element in a list
    - filter selects elements based on a predicate function
    - foldl and foldr accumulate results by applying a function over a list
- Understand the workings of the foldl and foldr functions.(Your text examples are in Haskell, but both functions are present in Scheme and Racket.)
  - foldl - fold left
    - are folding functions (head of the list) that reduce a list to a single value by applying a function recursively across its elements
  - foldr - fold right
    - starts from the right (end of the list) applying the function between each element and the accumulator from right to left
- Slides walk through construction of the implode, string2int, powerset functions. do look at how the process is done, how higher order functions are used to customize what the generic functions (map, fold, etc) are doing.
  - implode
    - converts a list of characters into a string
  - string2int
    - converts a string representation of a number to an integer
    - use higher order functions like map to process each char and foldl to aggregate the result
  - powerset
    - generates the power set (set of all subsets) of a given set

- Pay attention to section 8.5, analysis; this summarizes the main points of the chapter without getting into the details of code.
    - Higher-order functions capture common, typically recursive, programming patterns as functions
    - When HOFs are curried, they can be used to automatically define atomic functions—rendering the HOFs more powerful.
    - Curried HOFs help programmers define functions in a modular, succinct, and easily modifiable/reconfigurable fashion
    - In this style of programming, programs are composed of a series of concise function definitions that are defined through the application of (curried) HOFs
        - map
        - functional composition: o in ML and . in Haskell
        - foldl/foldr
    - Programming becomes essentially the process of creating composable building blocks and combining them like LEGO(R) bricks in creative ways to solve a problem.
    - The resulting programs are more concise, modular, and easily reconfigurable than programs where each individual function is defined literally (i.e., hardcoded)
    - The challenge and creativity require determining the appropriate level of granularity of the atomic functions, figuring out how to automatically define them using (built-in) HOFs, and then combining them using other HOFs into a program so that they work in concert
    - Resembles building a library or API more than an application program.
    - Focus is more on identifying, developing, and using the appropriate higher-order abstractions than on solving the target problem
    - Once the abstractions and essential elements have crystallized, solving the problem at hand is an afterthought.
    - The pay-off, of course, is that the resulting abstractions can be reused in different arrangements in new programs to solve future problems.
- Terms:
    - Partial application
        - the process of applying a function to some of its args producing another function that takes the remaining args
    - currying
        - transforming a multi-arg function into a series of single arg functions
        - helps with partial application
        - enabling functions to be reused with fewer args
    - higher-order functions
        - functions that take other functions as args or return functions as results

- first-class closures
    - closure is a function along with the environment in which it was created meaning it can capture variables from its surrounding scope
    - in a language where functions are also first class allowing them to be passed around freely like other variables

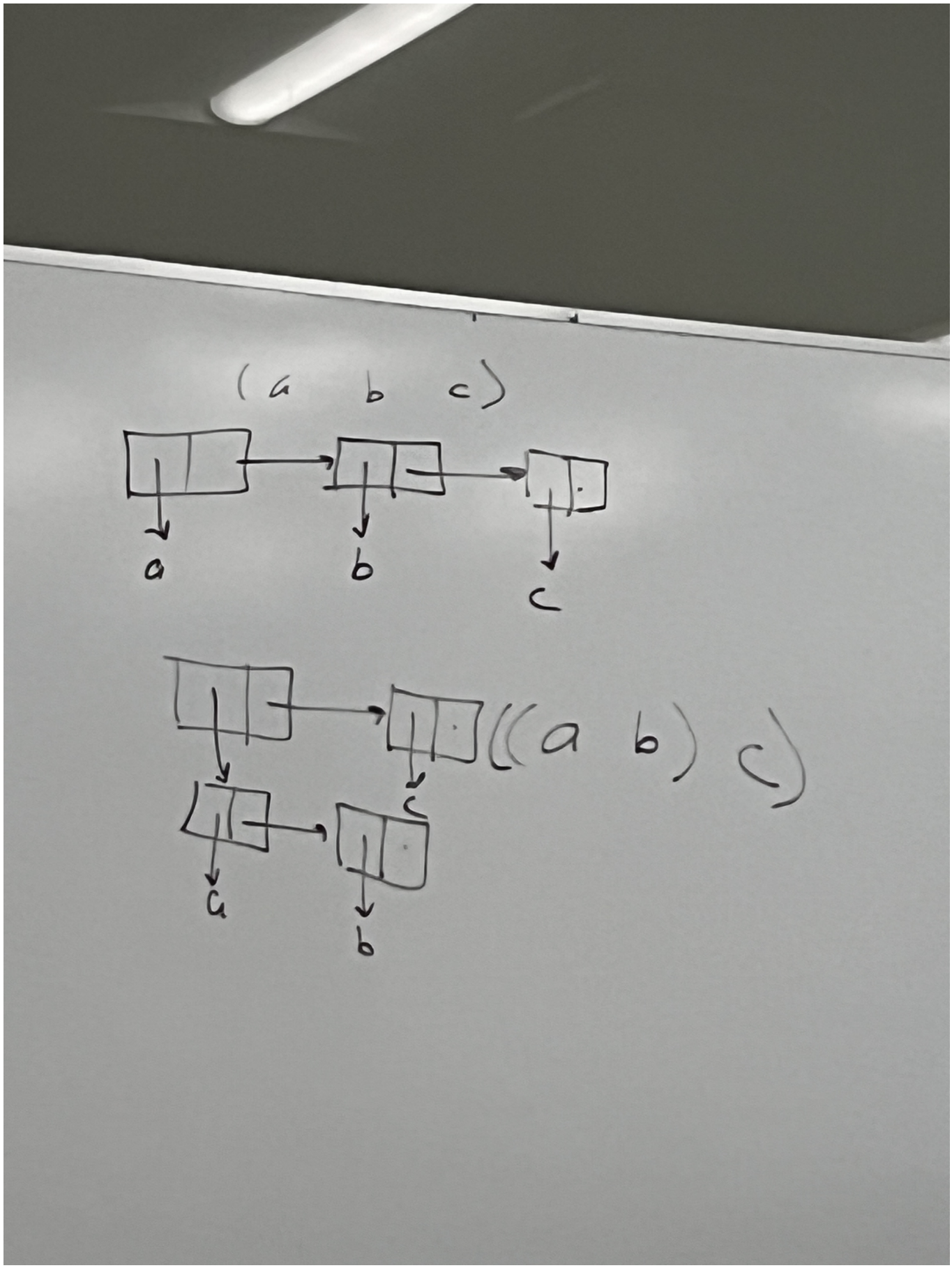
---

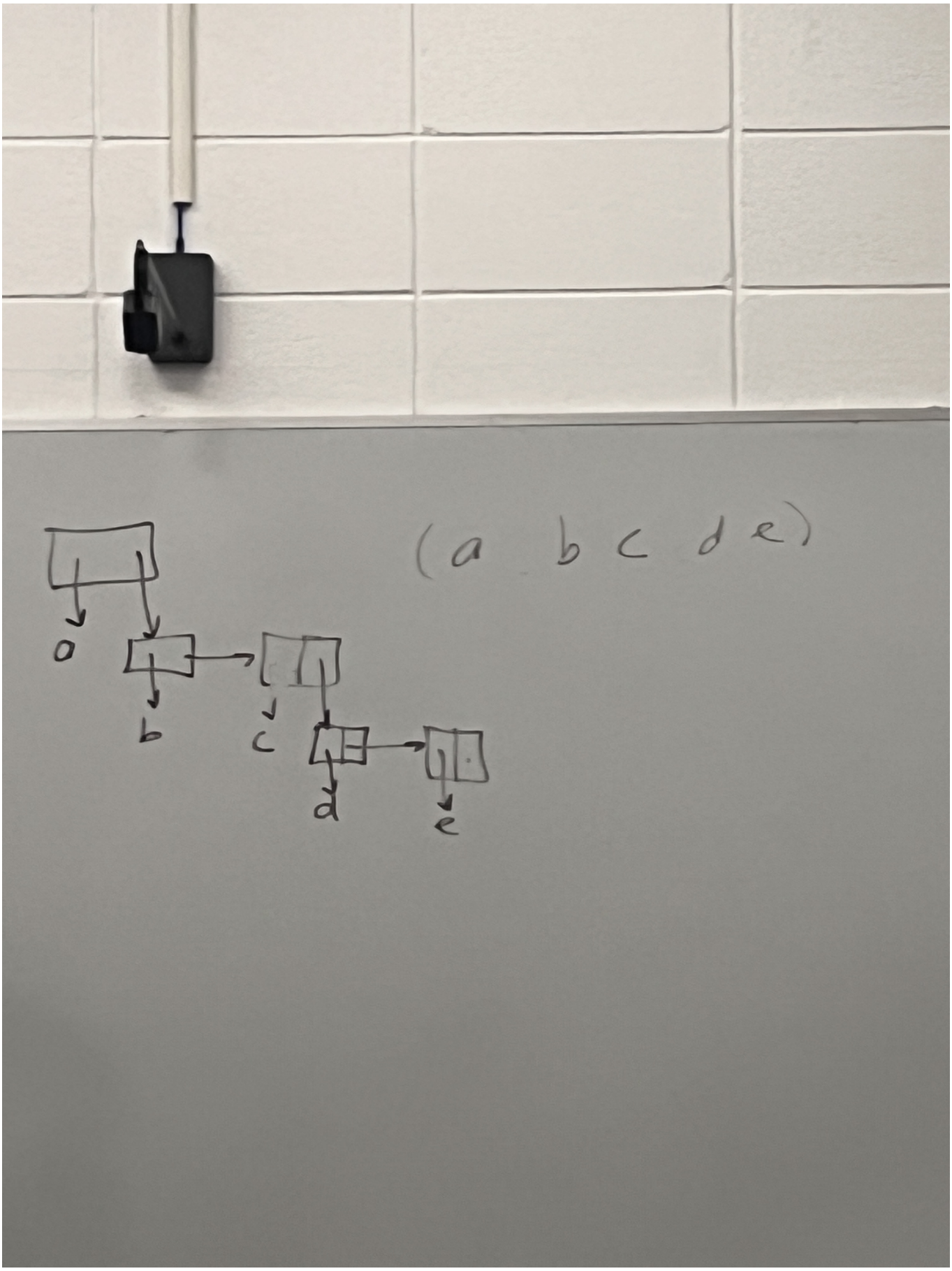
Chapter 9 Data abstraction

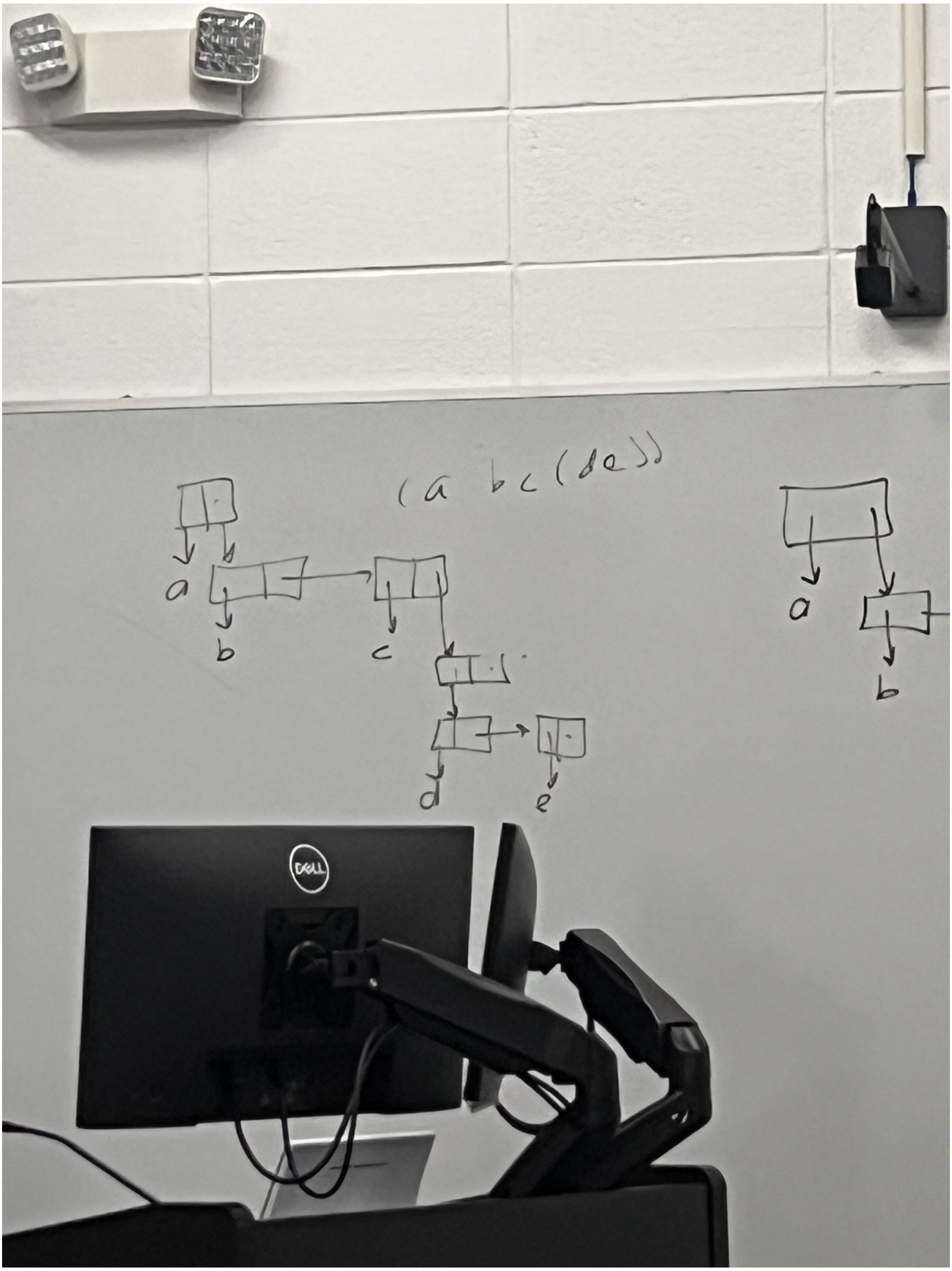
- What is the distinction between
    - discriminated union - tagged union
        - union type that carries an additional discriminator to indicate which variant of the union is being used at any given time
        - safer and allows compiler to enforce type checking
    - undiscriminated union?
        - union type without any tag or discriminator meaning that it can hold any one of a set of types
        - but theres no information to determine which one at runtime
        - programmer handles this which can led to errors
        - requires manual management
- What is a variant record? How is it usually implemented?
    - a data structure
    - combines both discriminated unions and records
    - it allows for different records to be stored in the same memory location depending on a tag
    - implemented using a discriminator which is an int or enumeration to keep track of the variant in use
    - union for the memory layout of diff values
- Explain the difference between
    - interface & implementation
        - interface
            - public facing part of a module or library
            - defines the set of functions method or types that are accessible to other parts of the program
            - specifies what the module can do
            - but now how it does it
        - implementation

        - the implementation is the internal logic of the module
        - provides actual code behavior behind the functions defined in the interface
        - ex. the actual body of a function or the internal structure of a class
    - Which does an application need to be aware of, and which can an application ignore?
        - application should be aware of the interface to know how to interact to know how to interact with a module
- Terms:
    - aggregate data types
        - these are data types that group multiple values into a single entity
        - arrays
        - records
        - tuples
    - inductive data types
        - these are types that are defined recursively
        - each instance of the data type is either a base case or a recursive case that builds on simpler instances
    - abstract syntax
        - that is the high level structured representation of code
        - focusing on the logical structure rather than the exact textual representation
    - type system
        - refers to governing types in a programming language
        - type system enforces constraints on what kinds of values can be used in various parts of the program
    - AST - abstract syntax tree
        - tree representation of the abstract syntax of code
        - each node represents a construct
        - used in compilers and interpreters to analyze and transform code

---


(a b c)
a
b
c
((a b) c)
a
b


(a b c d e)
a
b
c
d
e


(a b c (d e))
a
b
c
d
e
a
b