CS 131 Notes

Lecture Notes

1/9 - Week 1

- Language decisions. Need to minimize the cost of a project by selecting the best language.
- Costs associated with the language
    - Reliability: If we run the program in this language, how likely is it to succeed.
    - Training: How quickly can we get people up to speed in this language.
    - Maintenance: Is the documentation good and are there maintainers?
    - Program development: Is the language well suited for the task?
    - Runtime performance (lowkey doesn't matter): Do the programs run fast?
- Language design issues
- Orthogonality: Every axis is independent and you can make a choice on one axis without worrying what you chose on the other axis.
    - C has functions that return values of particular types. If you choose type T and create a function that returns a variable of that type. The choices you're making: The type T and the function that you create. This is **not orthogonal** because there are some types for which you cannot return it from a function. "For example, C won't let you define functions that return arrays".
    - You would rather have a language that is orthogonal, since it doesn't have these edge restrictions and exception cases and it is a whole lot simpler as a result.
- Efficiency: How efficient is your program when it is running? The main mark of efficiency is **time**, which also makes distinctions between real time and CPU time. There is efficiency in terms of **memory**, as well as the use of RAM/cache/secondary storage, etc. There is also **network** efficiency that deals with latency and throughput associated with the program consulting some server. There is also **power/energy** efficiency. We want languages with high efficiency and minimizes these quantities.
- Simplicity and Convenience
- Safety: Compile time checking or runtime checking or no checking. Can sometimes conflict with other goals. If you have runtime checking, then that could hurt efficiency. If you have compile time checking, then that could increase complexity and decrease simplicity
- Abstraction: How easy is it in your language to create a simple and abstract view of your program without having to go into details. OOP uses classes as a fundamental abstraction.
- Exceptions: Have a language that makes it easy to deal with errors, faults, etc. The language makes it clear when these unusual cases occur. Not everyone agrees on what should be exceptions though.
- Concurrency: Deal well with multiple programs running

- Mutability: How well does the language support changes to your program or to the language itself. Successful languages evolve. Want to make changes to the language without breaking previous code. You want to go for languages that will encourage evolving.
    - C has evolved from a language that previously was big on pointers and dereferencing them, but now you have to be careful since getting values from memory is slow?

1/11 - Week 1

- You can change the syntax of your language by defining new macros that change how you write code in that language.
    - A way of extending the language.
- You want to have a language that supports extension, and this is something that helps mutability.
- Syntax is where you have rules plus a parser.
    - The North Campus definition is “form independent of meaning”
- Colorless green ideas sleep furiously is an example where the syntax is right and the semantics are pretty off.
- Ireland has leprechauns galore is where the syntax is off, but you can understand the meaning.
- Time flies is where you have the syntax and semantics being correct, but having ambiguity in terms of how you parse the sentence.
    - So you have to be careful that your parser doesn’t have multiple ways of understanding the same syntax.
- Issues with syntax
    - Simplicity: Want syntax to be simple. You want a small number of rules and have the parser be easy to write.
    - Inertia: Want something that people are used to.
    - Readability: How long does it take you to read a program and figure out what it does.
    - Writability: How easy is it to program in this language.
    - Redundancy: How redundant is your syntax, and is that something you want or don’t want? Sometimes redundancy increases the reliability of your code and is thus something you want.
        - Small amount of redundancy can help your language design
    - Ambiguity: Don’t want syntax to be ambiguous because the program can mean different things on different machines. Cannot have an expression where two interpretations are correct.
- There is a fight between simplicity and readability since something might be simple but confusing as shit to read and understand
    - Postfix notation is simple but not readable.
- Scripting language push writability but at the expense at some other factors.

- Tokens are the primitives that you need to figure out before the rules and parser. They are the basic building blocks of your programming language.
    - Basically splitting up your line of code into atomic objects.
    - “Int main (void){ return getchar() }”
    - When you feed that in, the scanner will look at the individual bytes of your program, and converts it into a set of tokens (int = 9, main = 4, (= 97, etc). Internally all of these atomic units are numbers, and they will be the tokens.
- We can define whitespace as everything outside of the tokens.
    - Space, newline, tab, etc
    - They’re ignored and discarded as part of tokenization.
- Comments are also ignored and discarded.
- Tokenization is greedy which means it’s trying to find the longest token that it can find, going from left to right.
    - Iffy is a token in it of itself and we don’t tokenize just the if part.
    - Int iffy; -> INT, ID, ;
    - Int if: -> INT, IF, ;
- **Keywords** have a special meaning in a language, and are part of the syntax.
- **Reserved words** are words that cannot be used as identifiers (variables, functions, etc.), because they are reserved by the language.
- Reserved words violate the mutability principle since it’s not possible to add new keywords since that would break a lot of previous code.
- Once the compiler has its set of tokens, it will feed the tokens that the scanner found and gives it to the parser which will generate a parse tree.
    - The point of the parse tree is to give structure to our tokens. If you look at the fringe or the leaves of the trees, you’ll see all of the tokens that you started off with, but you have more information now because of the tree.
- A grammar specifies what type of parse trees you can create from a set of tokens.
- Context free grammar
    - Start symbol: The root
    - Finite set of non-terminal symbols: It will be the interior nodes of the tree. Basically all the nodes on a path from the root to a leaf will be non-terminal except for the leaf itself.
    - Finite set of terminal symbols: Aka tokens or the leaves
    - Finite set of production rules: Each looks like: a non-terminal symbol -> finite sequence of symbols.
- Example Grammar
    - Start = expr
    - Nonterminal = {expr}
    - Terminal = {Num, ID, *, +, (,)}
    - Rules
        - Expr -> Expr + Expr
        - Expr -> Expr * Expr
        - Expr -> ID

        - Expr -> Num
        - Expr -> (Expr)
    - Ex: Try to create the parse tree for a + b * 3
- If you want to prove that your line of code is correct, then it has to be separated into tokens and you have to create a valid parse tree according to the grammar that you're given. And that grammar contains the rules that allow you to create the tree.
- The example grammar is ambiguous because you could have multiple different parses because of the addition and multiplication, so we need to figure out another rule so that we can avoid getting two different trees.

1/12 - Week 1

- OCaml is a functional programming language, meaning that functions are first class objects.
- OCaml is a strongly typed language means that a function that's expecting a certain type won't allow you to really pass in something else.
    - Cannot pass in a double when an int is expected.
- OCaml is also able to infer types.
- Side effect is a change that you make to the program that isn't visible to the output.
- Types in OCaml: Int, Bool, Float, String
- Need to make sure that both your types are the same (int-int or double-double) when working with arithmetic operations.
- Type variables are when OCaml doesn't know the type of an argument.
- Fixed point refers to functions and inputs where f(x) = f(f(x))
    - f(x) = x mod 10
    - f 3 would be a fixed point because f(3) = f(f(3)) = 0
- Periodic point is where only f^n(x) = x with period n
    - "Apply a function n times to an argument and then at some point it will return x back to you".
    - This will happen every n times.
- No side effects means that you cannot have a loop, but instead you need to use a recursive function.
- List elements must all be of the same type. Lists are also immutable once you create them.
- Cons operator will add a value to the front of the list.
    - The execution order is right to left when you're using cons to build your list.
- You can use @ to append the elements from one list to another list but need to make sure that they have the same types.
    - X @ Y puts the elements of X first and adds the elements of Y next.

1/16 - Week 2

- Grammars are descriptions of languages and they are almost like programs.
    - Can be used to generate sentences.

- S -> a S and S -> b is a program to generate sentences, and in this case it would generate a bunch of a's ending with a b.
    - Nondeterministic program
    - Capital letters refer to nonterminal symbols
- A recognizer/parser needs to be able to take in the sentence and create the tree and determine whether the sentence is a member of the language.
- A grammar can go wrong if we have S -> a T and S -> b. So that means you can only have 2 possible sentences, and you're stuck. It is a perfectly valid grammar though.
    - The issue we have is when you use a nonterminal that isn't defined. **Useless rule**
- Another possible issue is having a nonterminal that is defined but not used. Not necessarily a bug but kinda useless. **Useless nonterminal**
- Another issue to look for is a nonterminal that is not reachable from start symbol. Basically the nonterminals never get called.
- Another issue is detecting infinite loops which is shown by S -> T and T -> Ua and U -> Tb. The last 2 rules are reachable, all the nonterminals are used/defined, but the grammar still doesn't work because if you execute it, you'll go into an infinite loop.
- Another issue is that you have nonterminals that can never actually produce a string a terminals.
    - Problem 4 is a generalization of problem 1.
- Last issue is trying to use grammar to capture constraints that are best captured elsewhere.
- We can create a grammar for identifiers. One of the rules in the C++ grammar is defining what identifiers can be named as.
- We don't use grammars to specify how tokens are created, but we apply the rules after the tokenization has been done.
    - Using grammars to specify details that are actually handled better by simpler technologies is a problem.
- The 7th problem with grammars is the problem with ambiguous grammars.
- Example of simple grammar that's ambiguous. You can basically write a single sentence that has two different parses which means you'll have multiple different answers.
    - E -> E + E and E -> E * E and E -> ID and E -> num and E -> (E)
    - Num + ID + ID can be parsed in two different ways.
    - The first 2 rules, especially **E -> E + E** because it has a problem with associativity, are the giveaways that the grammar can possibly be ambiguous.
- Discovering and fixing ambiguity are two different problems.
    - Discovering: Figuring out an input string that creates two different parse trees.
    - Fixing: The problem is that the grammar is too generous and we want to reject some parses that are currently allowed.
        - Look at the parses we want and what we don't want and figure out a rule that implements that.
        - Solution to the other grammar (where the problem is associativity) is E -> E + T and E -> E * T and E -> T and T -> ID and T -> num and T -> (E)

        - The grammar is now more complicated since we have another nonterminal but now it's better, but there's another problem that precedence rules aren't followed (additions on the left will be done before multiplications)
        - The grammar now says that addition is left associative.
    - To fix that problem (which is precedence), the solution is E -> E + T and E -> T and T -> T * F and T -> F and F -> ID and F -> num and F -> (E)
- Statements in C are ; and return; and return expr; and expr; and break; and continue; and { stmtlist }
- While (expr) stmt needs the parenthesis while in do stmt while (expr); doesn't need the parenthesis because of the semicolon.
- Dangling else is a problem of ambiguous grammar.
    - Example grammar: while (expr) stmt and do stmt while (expr); and if (expr) stmt and if (expr) stmt else stmt and switch (expr) stmt
    - If (1) if (0) f(); else g();
    - Solution is change to if (expr) tstmt else stmt
- Multiple notations for grammars.
    - BNF (Backus Normal Form) is where each grammar rule is nonterminal -> sequence of symbols which can be nonterminals or terminals.
    - EBNF is extended BNF where X* is zero or more Xs in a row and X+ is one or more Xs in a row. They can be written in BNF if we wanted to.
- Context sensitive grammars are where you have nonterminal symbols that depend on what's around it. Ex) Sb -> aS
- Regular inside of CFG inside of CSENSG
- Rewrite msg_id = "<" word *("."word) "@" atom *("."atom)">" in BNF
    - msg_id = "<" word dotwords "@" atom dotatoms">"
    - Dotwords = nothing or dotwords = "." word dotwords
    - Dotatoms = nothing or dotatoms = "." atom dotatoms
- The basic idea behind EBNF rules is provide a concise rule in place of a bunch of BNF rules. BNF is simpler but takes more rules to explain.
- Single EBNF rule: word = atom / quote string
- Translated to 2 BNF rules: word = atom and word = quote string
- Another EBNF rule: atom = 1*<any Char except specials, space, and ctls>
- quote string = <"> *(qtext / quoted pair) <">
- Qtext = <any char except carriage return>
- Quoted pair = "\"
- ISO standard for EBNF has operators X* for repetition, X-Y for except, X,Y for concatenate, X|Y for or.
    - List is in decreasing precedence.

1/18 - Week 2

- BNF is low level while EBNF is more high level.

- EBNF can works with a syntax diagram, or a directed graph.
- Example
    - BNF: Xs-> X Xs and Xs -> []
    - EBNF: X*
    - EBNF Diagram: Shows X with a loop
- Typically see diagrams being used in large grammars.
- Scheme syntax for conditional expressions.
    - <cond> -> (cond <cond clause>+)
      | (cond <cond clause>* (else <sequence>))
        - Trailing plus means one or more cond clauses
- Parsers will tell you why a particular statement is a part of a grammar.
- parse_cond() {

```
scanfor("(");
scanfor("cond");
If (lookahead(["(", "else"])){
        scanfor("("); scanfor("else");
        parsesequence();
        scanfor(")");
} else{
        While ()
}
```

  }
- These diagrams can represent finite state machines that can turn into code.
- Parse_cond has to be a recursive function because they have to be able to handle deeply nested parentheses.
    - So really what we're coding is a finite state machine + stack, which is called a pushdown automaton.
- Some grammars that don't need a stack if you can take your grammar and create a large ass syntax chart that only has terminal symbols.
- Homework #2 - Build a parser for arbitrary BNF graph
    - Have to support recursion
    - Have to support alternation - OR. Basically the possibilities that come with ABs -> [] and ABs -> a ABs and ABs -> b ABs. You may have to try all 3 of them.
    - Have to parse concatenation - Call a function corresponding to each NT in sequence.
- Let's say you write a parser and it works and the original grammar
- Expressions and their problems
    - Precedence - Handled by grammars
    - Associativity - Handled by grammars
    - User defined operators
    - User defined syntax
    - Side effects in expressions
- Prolog lets you define your own operators.

    - Op (500, yfx, [+, -])
    - Op (400, yfx, [*, /])
        - F is a binary operator that is left associative.
    - Op (200, fy, [+, -])
    - Op (700, xfx, [=, /=, -=, >=])
- Motivation behind functional programming
    - Clarity in that we have centuries of experience with this style.
        - Assignment isn't really natural to us. X = x + 1 should be false technically. In C and C++, there are problems with order of evaluation, but not really there in functional programming.
    - Performance because C and C++ have issues where you're only executing one thing at a time. Instead we want to escape from von Neumann bottleneck which is the bottleneck that comes with loads and stores from memory to CPU and vice versa. With functional programming, we can have the illusion of multiple CPUs.
- A function is a mapping from a domain to a range.
    - Partial function is where the domain is not entirely covered.
    - Given the same input, you need to have the same output.
- Functional forms are functions whose domain or range includes functions.
- No assignment statements and no side effects in functional programming.
    - A functional program is often part of larger systems that do have side effects. So the functions are functional but the system does more?

1/19 - Week 2

- Parser generator takes in a start symbol and a grammar, and returns you a program that can take in a sample program text and return you an abstract syntax tree.
- Custom types in Ocaml.
    - Type awksubNT = | Expr | Term … Each of the Expr, Term, is called a type constructor for awksubNT
    - The option type has two type constructors, None and Some _ which creates an option of type _.
- Type optionalInt =
  | Something of Int
  | Nothing
- Type (‘a) bintree =
  | Node of ‘a bintree * ‘a bintree
  | Leaf of ‘a
- Let rec preorder bt = match bt with
  | Node (v, lt, rt) -> v::((preorder lt) @ (preorder rt))
  | Leaf -> []
- Type (‘a) myList
  | cons of ‘a * ‘a myList
  | Empty

- Let rec sumList a = match a with
  | Empty -> 0
  | Cons (v, tail) = v + sumList tail
- Tail recursion is when you call the recursive function as the very last thing that you write.

1/23 - Week 3

- When thinking about implementing a language, we need to think about:
    - Where is the code running?
    - How is it going to be developed?
    - Implications of using a batch environment or interactive environment.
        - Read/eval/print loop in those interactive environments
    - Use on real time or embedded devices?
    - Distributed/web apps
- Compiler will take your program and translate it into machine code (low level code that has the same effect as that high level instruction) while an interpreter takes program and stores in text form in memory, and goes line by line to determine what to actually execute at each line.
    - Compiler: Translate into machine code
    - Interpreter: Keeps source code in RAM, execute it directly
- Interpreters will almost always be slower. However, the pros are that they are portable since you can run the same instructions as whatever machine you're using. It's also really simple to write and read. You also will have better error checking because it will catch errors as they happen.
    - It will slow the interpreter down a little with that index checking, but at least you're catching bugs.
    - Interpreters better with cross platform programs
- An error that is better caught by an interpreter than by a compiler is when you have an index in an array that is out of range.
- How can we get both the high performance of compilers with the bug catching features of an interpreter?
    - Use interpreter while developing and use compiler for production but there is a problem between if the two will output the same results for the same program.
- To improve efficiency, the interpreter can keep a stripped down version (no comments) of the source code, and then turn it into byte code which is like a compromise between just the code in plain text and machine code.
- JIT compiler reads byte code as its source, and convert to machine code.
    - A job of that compiler is to be able to determine which methods are being used a lot and which are not.
- Every time you start up a Java program, that JIT compiler starts from scratch.
- The number of possible compilations will increase exponentially with the number of arguments.
- Javascript also uses byte and then machine code.

- We have the problem of translating text into machine code. Software tools to help build programs include using assembler, linker, etc.
- 3 major types of programming languages
    - Imperative (procedural) - follow commands and sequence them C1; C2; … And also use variables and make assignments.
    - Functional - functions are the unit and you hook stuff together using function calls f(x) and you don't use any assignment statements. Never change the value of the variable. Variables never vary in a functional language.
    - Logic - Predicates are the units of computation. They are statements that are true or false. The syntax is p if q. No assignments and no function calls.
- Ocaml
    - Ocaml is an ML variant. It has static type checking to make sure that you're not calling functions with the wrong types.
        - Static type checking (checking types at compile time) makes your code more reliable.
        - This is like C++ and Java but unlike Python and Javascript
    - You don't have to write the types down in Ocaml though because it does type inference.
        - This is like Python and Javascript but unlike C++ and Java.
    - Ocaml also has a garbage collector
        - This is like Python and Javascript and Javascript, but unlike C and C++ cuz you gotta deal with your own memory.
        - Typically though, the automatic one will have worse performance than a manual collector given that that programmer knows how to use it.
    - Good support for higher order functions.
        - This is like Python a little, but unlike everything else.
    - Lists are homogenous and can have any length, while tuples can contain any types
    - Variables like x and y match anything and bind variables. _ matches everything but no binding. P,Q matches 2-tuple, P::Q matches a non-empty list where P will take on the head value and Q will take on the tail.

1/25 - Week 3

- cons(x,y) = x::y
    - The type is 'a * 'a list -> 'a list
- car(x::_) = x
    - The type is 'a list -> 'a
    - But this technically doesn't cover all the cases. The empty list doesn't work. The pattern matching only matches non empty lists.
- The real versions of car and cons
    - Let cons = fun t -> match t with | (x,y) -> x::y
    - Let car = fun l -> match l with [] -> [] | (x::_) -> x

- The core components of Ocaml include lambda expressions where fun x -> E and match expressions where match E with pattern 1 -> subexpr 1 | pattern 2 -> subexpr 2
- Currying is taking a function with multiple arguments and turning it into a higher order function with just one argument.
- A curried version of cons is
    - Let cons x y = x::y;;
        - The type is ‘a -> ‘a list -> ‘a list
- The advantage of currying is that you don’t have to decide all the arguments to a function right away.
    - You can define another function that has some specific functionality of another function.
        - Let cons1 = cons 1;
        - This function always prepends one, so you can just call cons1 with one list argument.
        - If we did it in the tuple way, it would look like
            - Let cons1 y = cons (1,y)
- Using the tuple is like the data oriented way of looking at things, but in functional programming you should think of functions as being able to used in that way as well.
- Real talk though, the compiler does save your ass sometimes and saves a bunch of debugging time with the statically typed characteristic warning you about type mismatches.
- If you use the function keyword, you have to just have pattern matching because you’ve implicitly passed in an argument that you need to be able to parse through.
    - Use function whenever you have a nontrivial problem and you want to do pattern matching.
- Fun x -> x + 1 and function x -> x + 1 are technically equal to each other but for that type of trivial expression, it’s always better to use fun.
- Use fun when you want to create a function that does not have a name. Normally, you want to do that when you trying to compute a subexpression that’s maybe part of the result of another expression.
- Function application is left associative, so if we type in f x y, then we assume (f x) y.
- Minlist - 2 different ways
    - This will just work with ints
        - Let rec minlist = function [] -> 99 | h::t -> let m = minlist t in if h < m then h else m
        - Type is int list -> int
    - This should be general
        - Let rec minlista inf lt = function [] -> inf | h::t -> let m = minlista inf lt t in if lt h m then h else m
        - Type is ‘a -> (‘a -> ‘a -> boolean) -> ‘a list -> ‘a
- Types
    - A type tells us a set of values.
        - For example, an int tells us we can have 5, -4, etc

    - Tells us what you can do with them. What kinds of operations you can have on those values.
- Primitive types are those that are built into the language. (The values and operations)
    - They cannot be deconstructed in simpler types as well.
- Constructed types are those that are supplied by the users. An example is an array.
- There are some issues with portability. For example, if we decide the int type can go from -2^31 to 2^31 - 1, then it might not actually work on 16 bit machines.
- IEEE-754 floating point type
    - 2^-127 or something
    - Infinity and Nan
    - Operations: Add, subtract, multiply, divide, sqrt, round to nearest representation
- Hardware can trap on overflow, underflow, NaN generation, or inexact result (rounding error).
- When designing a type, you need to decide what happens in the strange cases
    - Portability problems, exceptions, traps, etc

1/26 - Week 3

- For the list [“3”, “+”, “4”], the prefix is [“3”, “+”, “4”] and the suffix is the empty list
- An acceptor is a function that takes a derivation and a suffix as arguments. It will return Some (d’, s’) or None
    - The type of the function is ‘a -> ‘b -> ‘a * ‘b Option
- A matcher is a function that takes an acceptor and a fragment as arguments.
    - The type of the function
    - 1) Finds the next matching prefix for the given fragment
        - Find the leftmost derivation where the terminals that you can back make up a prefix of the fragment.
    - 2) If there are no matching prefixes, returns None
    - 3) Calls the acceptor with the suffix of the fragment and the derivation
    - 4) If accept call returns None, then loop else return accept call
- Homework 3 deals with sequential consistency.
- Java doesn’t have pointers, has garbage collection, has mutable state, and is object oriented.
- Java types
    - Int
    - Byte -128 to 127
    - Char - unicode
    - Short -2^15 to 2^15 - 1
    - Int -2^31 to 2^31 - 1
- Java arrays
    - Int [] i;
    - I = new int[5];
        - New dynamically allocates memory

- If you have some class A, and you want to create an object of that instance, then you need to do A a = new A();
- Classes can inherit from other classes.
- Main methods need to be written inside of classes. The class has to be written in a file called className.java.
- Only one public class per file
- public static main (string[] args){}
- Object is an instantiation of a class
    - Objects only exist at runtime, when you've actually created your objects.
    - Classes define the structure of the object.
- Need to remove all .class files.

1/30 - Week 4

- Types are like rules, they are annotations that are useful to the programmers as well as to the compilers, since they are able to create optimizations if they know the type of a variable beforehand.
- Compiler and programmer can also infer types from other information.
- Static vs dynamic type checking
    - Static is more reliable since once we've compiled the program, we know 100% sure that we won't have any errors
    - Dynamic allows you to be more flexible and simple.
- Strongly typed language is one where the type checking is good enough to guarantee that there won't be any type errors when the program runs.
    - They are a bit less flexible though.
- C++ uses static type checking, but not strongly typed because you can use pointers.
    - Char buf[100] and then void *p = buf and then int *ip = p and then p[0] = 26;
- Structural equivalence says that two types are the same if their layout and their operations are the same.
- Name equivalence is when two types are the same if they have the same name.
- Structural requires you to look and see how the types are implemented and what operations you can do on the type.
- A language can use both name and structural equivalence.
- These two structs are the same structurally, but not name wise
    - Struct s{ int val; struct s *next} and Struct t{ int val; struct t *next}
    - Struct s a; and Struct t b;
    - A = b;
    - One caveat is that if one of the next got changed to pext, then structural equivalence would be lost.
- C won't let the above happen since it uses name equivalence in this case
    - Typedef int t1; and Typedef int t2;
    - T1 a; and T2 b;
    - A = b;

- C is okay with the above.
- If class A is a subclass of type B, it is a subtype.
- A = b should be allowed if B's type is a subtype of A's type.
    - Equivalent to tryna to determine if int subset of long (b subset of a)
- Int const *pi = &n means I am pointing to something and I promise not to change that character using this pointer.
- A coercion happens when you're doing an assignment, the types don't match, but the system is like nah Imma put this conversion in for you during runtime.
    - Extra code is needed to handle that conversion but if the types and subtypes actually fit, there's no operation needed during runtime.
- Types are not just sets of values but also contain the operations that they can do.
- Subclass inherits all the operations of the parent class and adds some other operations.
    - TODO difference between char * and char const *
- Polymorphism says that the function f(x) could have multiple implementations based on what the type of x is.
    - The code that calls the function looks the same, but the implementation is diff
- When you have a function that is overloaded, you identify the implementation by looking at the argument types.
    - You can also overload based on the result type.
    - Double x = sin(10) will probably use the double -> double implementation
- Name mangling is when you can specify different implementations of function f by calling them f_1, f_2, etc in the symbol table so that it knows which implementation to use.
    - All the compilers need to have the same name mangling function.
    - At compile time, the compiler will look at the arguments, and knows which one it is going to call.
- Overloading and coercion are kind of ad hoc polymorphism.
- Universal polymorphism is where you have an infinite number of types where the programmer can specify.
- Parametric polymorphism is when a function's type contains a type variable.
    - 'A list -> int is an example because a can really be any type.
- If the compiler doesn't know what the type of the object might be, then sometimes a runtime cast might be needed.
- With templates, the compiler compiles the code for each instantiation, but with generics, there is just one copy of compiled code. But with templates, each copy is type checked, but it's harder to check something generically.
- Java Generics and subtypes
    - List<Integer> l = new LinkedList <Integer> ();
    - l.add(new Integer(0));;
    - Integer n = l.iterator().next()
- Because the set of operations you can do is not a superset, it is not a subtype.
- The idea is that in order for something to be a subtype s of another type t, s has to be able to do all the operations that t can do. If it cannot, then we can't do List<Object> lo =

List<String> ls even though String is a subtype of Object. The fact that we're making a list is what messes things up.
    - Not allowed to convert list/collection of Strings to list/collection of Objects.
- You can also put the wildcard ? in place of type names
    - Collection <?> c in your argument will take a collection of anything.
- Collection <Triangle> is not a subtype of Collection <Shape> even though Triangle is a subtype of Shape.
- Bounded wildcard is a ? with a constraint. Collection <? Extends Shape> means that the thing you pass in has to be Shape or a subtype of Shape.
- Can also have bounded type variables. Void <T extends Shape> print(Collection<T>)
- Super constraint is the opposite of extends.
- We need to place constraints on parameterized types in Java.
- Ducktyping is a style of type checking whether you don't ask the object what type it is, but you just care about if it can do the operation you want it to do.
    - If the methods (quack, waddle) work, then why tf do you care if it's a duck or not.
    - The downside to this approach is that you'll have a shit ton of runtime errors.
    - But honestly you get some great flexibility and simplicity.

2/1 - Week 4

- Java's original design was based on UNIX. Started from the Unix kernel © and apps would be written in C.
    - One of the problems is that the embedded world is not interested in high end CPU (you want the cheapest CPU to put in your toaster). And thus, there comes the problem of heterogeneous CPUs. Everybody wants to use their own unique CPUs.
    - Solution is that you write your code in C, and then compile for different architectures so you have multiple copies.
        - One version that runs on ARM, one for x86
    - This works, but it's not the most optimal because you need to test N different versions.
- Another problem is that machine code is bloated and at that time we had poor network speed and thus it would take a lot of time to send the executables.
- Another problem is that it's not object oriented, so maybe we should switch to C++, but it has the recompile the world problem, since you you change one little line, then you have to recompile the whole file. Plus, C++ is hella complicated and crashes too often, and they have problems of unportable behaviour.
- A solution to the complicated problem was going with the Smalltalk language, which was object oriented and simple. It also compiled the source code into machine independent bytecodes, and then to make it work on different machines you need to create bytecode interpreters, which were not too difficult to create. Smalltalk also had an automatic garbage collector, which was really helpful.

- Then, they took all the above ideas, and made a new platform that had a C like syntax to it plus network plus parallelism called Oak.
- Mosaic was the first successful browser, but it had a problem in that it was written in C++.
- So, now, the other people made a browser called HotJava did everything Mosaic did, but it didn't crash because it used Oak/Java. They also had applets, which are downloaded from the website to the browser (via bytecodes over network), and thus you could extend the functionality of your browser.
- Mosaic had their own version in the form of plugins, but didnt work as well.
- Applets had a problem in that it was a good match for servers but not really for clients, since there was a lot of work in parallelism and was overengineered and heavyweight for the clients.
    - So you run Java on the server side but not on the client side.
- Types in Java: Primitive and Reference types
    - Primitive: Byte (like char in C), char, short, int, long, float, double, boolean
- There's no address of operator (&x) in Java.
- In C++, you can have different machines that have different bit sizes for ints, longs, etc. But in Java, these values (32 for int, 64 for long, etc), these are hardwired down. All the machines have to abide by IEEE format. Java is very big on portability.
- You can think of reference objects as values that point to places in memory.
    - When you assign something in Java, you're copying a pointer, you're not dealing with the actual 1 MB data object, which you would be dealing with in C.
- Arrays in Java are references. Int [] foo = new int [100]; Java has a garbage collector so this array will always stay there and thus they are always heap allocated. They also have a fixed size once allocated. You can also return arrays from methods. Subscript checking when trying to access an array is required. Arrays are also initially zero when you create them.
- Java has classes and uses single inheritance - class inherits from just one parent. Thus the object hierarchy is a tree and the root of the tree is the class Object.
- The substitute for multiple inheritance is called an Interface.
    - Parent has m() and n(). The child gets those 2 methods and that saves work. With an interface, though, it's like you're inheriting a debt which means that if the child implements the interface (Let's say the inheritance declares o() and p()), then the child has to implement o() and p(). The interface placed a constraint on the child's behaviour.
- An abstract class is where you have some methods implemented but some that are declared but not defined.
    - Class p{ define m() and n(); abstract int foo() }
    - Basically supply initial code to the subclasses, but there are some holes for the child classes to define themselves.
    - This is kind of the combination between classes and an interface.
- When to use inheritance vs interfaces. Pick the approach that maximizes code sharing which is the basic goal.

- Collection is a general class (so you can have a collection of anything) and its children are List, Set, and AbstractCollection which is an abstract class.
- Cannot construct an instance of an abstract class nor an interface.
- Final is almost the opposite of abstract. Abstract is saying here is the API, but no implementation. Final has both the API and the implementation, but you cannot override. A child cannot override a parent's public final methodName().
- Final methods are used because we don't want children screwing up key methods.
- Erasure means that if you have a List<Integer>, we just say List.
- Java has a rule about methods. If the method can throw an Exception, you have to declare that in the definition.
    - Protected void finalize() throws Throwable.
    - Signal to the programmer that if you call finalize() then you better be prepared to put it in a try exception. You don't need to put it in the definition if you catch the exception within the method itself.
- Finalize gets called by the garbage collector just before an object is reclaimed.
    - Protected keyword is intended only for the garbage collector.
    - Useful for when you're interfacing with C++ code where you have explicit calls to the destructor
- Clone() gives you a copy of the object that calls it, but the object has two different locations, so they're technically 2 different objects. You can also however get a CloneNotSupportedException.
- Built in object called Thread in Java. Thread gets created by calling new Thread() and the resulting state is NEW (as opposed to RUNNING, STOPPED, etc) and then you can invoke the start method which allocates the OS resources, and then invokes the run methods, which is the code executing inside the thread, and now the state is RUNNABLE (which means the OS could run it if it wants to, but doesn't have to at that moment), and then the thread code can either run (result is RUNNABLE), sleep (result is TIMEDWAITING), I/O (result is BLOCKED), wait (result is WAITING), or exit the run method (result is TERMINATED).

2/2 - Week 4

- Print homeworks and solutions to homeworks.
- Tail recursive is where you only call the recursive function at the end. So, it's not recursive when you have something like h::recFunction() since you cannot add h until recFunction() returns a list. The alternative is to keep track of the updated list through an additional argument.
- A memory model defines necessary and sufficient conditions for knowing that writes to memory by other processors are visible to the current processor and writes by the current processor are visible to other processors.
- Strong memory model is where all processors see exactly the same value for any given memory location at all times.

    - Weaker memory models have instructions that are required to flush out the local cache in order to see writes made by other processors.
- A lot of times the compiler can make specific optimizations that speed up the code, but it can also mess up the timing of when some instructions happen, what effects they have, and what other threads view the current situation as.
- The JMM describes what behaviors are legal in multithreaded code, and how threads may interact through memory.
- Synchronization mostly means that only one thread will enter a synchronized block and no other thread can enter until the first one exits. Also memory writes by a thread before or during a synchronized block are made visible to other threads.

2/6 - Week 5

- Synchronization boils down to situations where one thread is writing and another thread is either reading or writing to that same variable.
    - The behavior depends on the type of the variable.
- If int, accesses to variables are atomic in that accessing o.v1 will always happen before/after another instruction, but if long or double, then result could be garbage.
- If v and w are long/double, when the Java compiler sees o.v = 0 and o.w = 3, if it wants to, it can partially set v and partially set w, and then go back and finish later.
- One of the mantras of Java is portability so that you can write once and run everywhere.
- While optimizing, Java can not only split loads/stores, but it can also reorder assignments as long as a single thread cannot tell the difference.
    - O.v = 2; o.w = 6; o.e = 4; can be done in the opposite order.
    - It would do this for performance reasons (storing in increasing memory location order is faster and more helpful for the cache layout).
- Code that does the above optimization is great. But you have to be careful in multi-threaded code in that o.w = True; o.y = 7; o.w = False is terrible.
- Every time a program accesses a volatile variable, the access has to be done in order and specified in source and thus loads/stores will come from memory. Volatile does not mean atomic. Atomic means that if someone is storing a new value to a variable, your access will either come before/after that store.
- Compiler cannot do as many optimizations if you have a lot of volatile variables.
- Synchronized code means that we have mutual exclusion, which means any block of code running on an object o locks out any other method running on that same object.
- The o.wait method will remove all the locks held by this thread and waits until the object o becomes available.
    - The idea is this will be called when a thread is waiting for an object it really wants and first removes all the other locks it has and then waits. This really helps in avoiding deadlock.
- The o.notify method will wake up one waiter thread. O.notifyall() is similar.

- Exchanger is a class where you have an Exchanger object x and two threads that want to exchange some information. Thread 1 can do x.change(v1) and get back v2 and Thread 2 can call x.change(v2) and get back v1.
- CountDownLatch is like a barrier where you have a bunch of threads that enter the latch and they all wait until all N of them are sitting in the latch, and then they all start.
- CyclicBarrier is a little different because the barrier is reusable. Like horses going around the track and once a loop is done, then waiting again for everyone else to arrive.
- Names are identifiers and in C, you cannot have “if” as an identifier. You have to determine which words are reserved, if the language is case sensitive, and what the possible character set is.
- Binding is a relation between identifier and what it stands for.
- In n = 10, n is not necessarily bound to an integer, but it is bound to an address.
- When thinking about what can be bound to a variable, it’s important to determine what the binding time is, or when the binding is created. Can be created at
    - Runtime, at every assignment or Compile-time, when program was statically analyzed or ABI (Application Binary Interface) define time or Link-time or Runtime at block entry (such as the variable i in a for loop)
- Namespace is a collection of names and their associated values. It’s a partial function.
    - No two symbols in the table are the same and thus it is a function.
- Primitive namespaces are built into languages.
- Basically you can use the identifier x if they are in different name spaces.

2/9 - Week 5

- Press semicolon when you want to see all the results
- In a valid rule, you cannot have more than one predicate or expression in LHS.

2/13 - Week 6

- Imperative languages (C++, Java) and Functional (Lisp, ML) and Logic (Prolog)
- With functional languages, there are no side effects and no assignment statements.
- With logic languages, we give those up and we give up on functions. We use predicates instead. As long as you’re talking about the truth of something, you can still say it.
- With logic programming, you want to split up into the logic (which is about correctness) and control. The latter does not affect whether it's the right algorithm, but is concerned with how efficient the implementation is.
    - The coupling is very difficult to do in imperative and functional.
    - We can focus on either aspect at different times.
    - First figure out what you want, then figure out how you want it to go fast.
- In order for sort(L, S) to be correct S has to be the sorted version of L.
    - S has to have the same elements as L and it has to be sorted.
    - sort(L, S) :- perm(L, S), sorted(S).
    - Now that we have to subproblems, we can solve each of them separately.

- Every time you write a clause (something ending in a period), the scope of the variables in that clause is just that clause.
- These are logical variables that can be instantiated with anything. You should think of the above statements as being **true for all L and for all S**.
- Let's examine sorted first
    - sorted([]). and sorted([_])
        - Means that every empty list and one element list is sorted.
    - Now we need to talk about the recursive list.
    - sorted([X, Y | L]) :- X =< Y, sorted([Y | L]).
        - X will be the first element, Y is the second, and L is the rest of the list.
        - Basically saying that X and Y needs to be in order and the rest of the list is sorted as well.
- Then, let's look at perm
    - perm([], [])
    - perm([X | L], S) :- perm(L, S1S2), append(S1, S2, S1S2), append(S1, [X | S2], S)
        - Order of the calls is very important since it affects the efficiency.
- Let's look at append
    - append([], L, L)
    - append([X | L], M, [X | LM]) :- append(L, M, LM)
- Let's do member
    - member(X, [X | L])
    - member(X, [_ | L]) :- member(X, L)
    - If you enter the query member(Q, [3, 2, 5]), then it will return you Q = 3 since that is the unification that makes the first fact true. If you enter semicolon, then it will run it again and return you other solutions like 2 and 5.
- Prolog is almost like a database system where you have information in the form of facts and rules, and then you can ask queries to that system.
    - When a query gets entered, Prolog will look for a clause where the head of the clause matches the goal. I think it will try facts before it tries rules.
        - If it matches a rule (instead of a fact), it will have to match the subgoal which is basically what is contained in the if statement.
- Fail is a special built in keyword that would always fail.
- A term in Prolog is an atom (any string), a number, or a variable (can become instantiated on success and unbound on failure) or a structure which is a tree structured object where the leaves are atoms, numbers, and variables.
- We can talk about functions in Prolog but we cannot execute them
- We have a predicate called is, and it has arity 2 and the first argument is a number and the second argument is an arithmetic expression.
    - is(4, '+'(2,2))
        - Equivalent to N is 2+2. It's just that the above is what it gets converted to.
    - is(N, '+'(2,2)) will output N = 4
- Is (1368, '*'(N,N)) is an error because they must be ground terms and not variables.
- Example of syntactic sugar in [3, 2, 5] being '.'(3, '.' (2, '.'(5, [])))

- We want first argument to be known (or at least the length is known)

2/15 - Week 6

- Facts are terms with no logical connectives.
    - prereq(CS32, CS131)
- Rules are like facts,but you can use logical connectives like the if connective and AND.
- Queries are things are you ask the database.
- Prolog execution involves some top level goal that also may be composed of some subgoals.
    - The execution is normally a huge AND-OR tree and you're telling the Prolog interpreter to go find an answer.
    - DFS is used left to right to try to find a viable answer.
- If you have a rule with two subgoals, then it won't look at the second until it is done answering the first.
- Prefer to have answers on the left hand side of the tree since that's where Prolog where check first.
- Need to make sure that the algorithm is correct and then worry about whether something is efficient or not. For the query "Repeat. Newline. Fail" then
    - Repeat. Repeat :- repeat. Will output an infinite number of new lines.
    - Badrepeat :- Badrepeat. Badrepeat will give you an infinite loop.
- To debug, you can put write predicates in your queries.
- There is a Prolog "debugger" by putting in the query trace.
- Unification is the way Prolog takes variables and determines what their values should be. Process of taking two two terms and making them equal by setting the values of the variables in a particular way. (aka binding a value)
    - Fails if there is no way to make the two terms the same.
- When you write a query with variables, the program will try to bind the goal variables to the program values we have already defined.
    - This is different because the caller includes variables in it. In normal programming, you'd never really pass in a variable to a function, you normally pass in a value.
- You can also write a query with constants/atoms as well.
- Unbound variables are represented with _number.
- When you have a variable, you can unify/bind it with a structure like f(Y).
- Circular terms is when you have a function symbol whose argument is the same structure.
    - Fact: e(X, X).
    - Goal: e(Y, f(Y))
    - X will bind with Y and also with f(Y). Then, when you try to print out the value of Y, you will get Y = f(f(f(f(....)))))
    - This can happen even if you remove the e predicate.
- Negation as failure in Prolog

- \+(x) :- x, !, fail
- \+(_)
- This basically means that anything I cannot prove must be false.

2/16 - Week 6

- Finite domain solver allows you to set up some constraints that need to be satisfied and allows you to ask for answers that satisfy the constraints.
- Useful functions
  - fd_domain(Vars, Lower, Upper)
    - Lower and Upper are ints that specify the lower and upper bounds, and Vars can either be a single variable or a list of variables.
    - This will basically take in the list, and make sure that all the values are between the upper and lower bounds
  - fd_labeling(Vars)
    - Succeeds with a unique solution to the constraints on all the elements in Vars
  - fd_all_different(Vars)
    - Constrains all the elements in Vars to be different.
- To specify a constraint, you say fd_domain(X, 0, 10)
- FdExpr1 #= FdExpr2 sets up a constraint that says that the first argument has to equal the second argument.
  - X + Y #= 5
  - Use this for both Finite Domain Solver and for the normal implementation.
- Cut operator is when you backtrack to a cut and you just end there. Aka all subgoals to the left of the cut get pruned, and all clauses below of the same predicate also get pruned (never explore them).
  - Cut will always succeed on it's first try, then it would go into the fail, and it will backtrack into the cut, and everything gets pruned to the left and all the other clauses of the same predicate get pruned.
  - neg(Goal) :- Goal, !, fail

2/20 - Week 7

- Closed world assumption is where anything I cannot prove must be false. If you have a list of all prerequisites, then you cannot say anything about a prereq that isn't in the list.
- 3 possibilities in Prolog is yes I can prove it, yes I can disprove it, or I don't know.
- Propositional logic
  - "It's cold today" = P or "The 405 is busy" = Q
  - Some people may say it's true or false. But we can assume it is either/or
- We give each of the propositions arbitrary names, and assume each is either true or false.
- We can also have connectives which let you build larger propositions out of smaller ones.

    - P ^ Q, P V Q, P -> Q
- The Prolog equivalent of AND is P, Q. The Prolog equivalent of OR is P; Q. The Prolog equivalent of implies is P :- Q
- P -> Q is equivalent to saying ~P V Q
- Tautology is a statement in propositional logic that is true regardless of what the values are for the elements that compose the statement.
    - These tautologies are redundant since they're independent of what the constituent values are.
- First order logic is called predicate calculus.
- An example of FOL
    - All men are mortal. Socrates is a man. Socrates is a mortal.
    - FOL says that the first 2 has to imply the 3rd. We cannot however prove this with propositional logic since there isn't a set of connectives we can use to represent this new statement.
- In FOL, propositions can have arguments.
    - "Freeway X is busy" = P(X) and "It was cold on day Y" = Q(Y)
- In a way, FOL is a generalization of propositional logic.
- We also have quantifiers "for all" as well as "there exists".
- An example of a tautology in FOL
    - For all X, (P(X) -> Q(X)) ^ P(a) -> Q(a)
    - For all X, (P(X)) <-> ~There exists Y ~P(Y)
- For something to be decidable, you can give the system any question, and eventually the system will give you a yes/no answer.
    - Euclidean geometry is decidable while integer arithmetic is not.
- We want to have a system that can decide as much as possible and we want it to succeed as much as possible.
- Clausal form is a simplified version of predicate calculus/FOL. Can take any set of predicate calculus statements and turn them into a set of equivalent clauses.
    - A clause is A1 ^ A2 …. Am where each Ai is a predicate, and if you know that is true, then you know that B1 V B2 V …. Bn.
- In our equivalent statements, we don't have any quantifiers and we just use special variables instead. The statements are logically equivalent to the original.
- Horn clause simplified assumptions by saying that n <= 1. 3 possibilities
    - If n=1, m=0, then you have a Prolog fact
        - man(Socrates)
    - If n=1, m > 0, then you have a Prolog rule
        - mortal(X) :- man(X)
    - If n=0, then you have a Prolog query
        - ?- mortal(Socrates)
        - We know that the expression reduces to False <- A1 ^ A2 …. Am
- Prolog will also go by truth by contradiction. Let's say you have your database of truth with a lot of propositions. Then you get some query. You assume that query is false, and

if you assume that, and you show that ~m, p, q, r, …. Is false, then the whole thing is true since you have found a contradiction.

- It uses the resolution principle to do this. It lets you take two true statements and then infer a 3rd true statements from the first 2.
- Simplified version for Horn clauses. Example:
    - H <- A1 ^ …. ^ Am is the clause in our database.
    - _ <- G1 ^ … Gn
- Prolog wants to be able to compute a match between G1 and H. We create a unification U.
    - The new goal list is A1U ^ ….. AmU ^ G2U ^ GnU
- This is called logical inference.
- Visibility modifiers in Java
    - Private means that the variable is only visible within the current class.
    - The default (just declaring int x) is visible in the current class and in another class in the same package.
    - Protected means that the variable is visible in the current class, in another class in the same package, and in other subclasses outside the package.
    - Public is all of the above plus elsewhere.
- Ocaml Namespaces
    - You basically have a struct that contains a bunch of declarations of variables/objects and maybe some implementations.
    - There are also signatures, which define some modules and could use something that the previous struct defines.
    - Functors are compile time functions from signatures to signatures
- You will have fine grain control over visibility using signatures when using Ocaml namespaces over Java.
- We need to manage storage for arrays, dynamically created objects, global variables, functions (the machine code for them), stack for recursion (pointers to other stack locations) and local variables and function arguments and flags and return address and buffers for I/O (like getchar() function), etc.
- The machine code that manages this memory is called the memory manager, which is also something else that needs to be stored somewhere.
- When dealing with arrays, Java does subscript checking, and it will do checks to make sure that the index is not negative or out of bounds. Basically you have two conditional branches.

2/22 - Week 7

- Array access normally is done through alpha + i * sizeof a[0] where alpha is the initial address of the array and i is the index. This is fine if the size is 2 or 4 or some other multiple because we can do shifts.

    - A problem occurs if you have an array of structs and each of the structs contains 3 chars (so 3 bytes). Then you'd have to multiply by 3, which isn't that efficient. The solution is to add padding so that each struct is 4 bytes.
- There is a tradeoff between memory and speed.
- Array slices are when you want to take some subset of an array and use that somewhere.
    - If you have double a[10][20] and you just want the 5th row, then you can pass it into function f by doing f($a[5]).
        - That will give you a row or a horizontal slice. You cannot do vertical slices.
- An array descriptor is something with 4 components, the first points to the first item of the array data, the second points to the lower bound, the third to the upper bound, and the fourth is the stride, which is the distance between the starts of array elements.
    - Stride is 8 for typical arrays of doubles. This stride allows you to increase it by a lot and skip over the whole length of the array so that you get the next element in the column.
- An associative array is where instead of the index to the array being an integer, the index is a string. An example is a dictionary in Python. If you were to implement those in C, you would need to use hashtables.
- To do a DOS attack on a web server, you can issues a bunch of bogus requests that you know won't be in the cache, and thus will always have to go to the main server to get the value and you will crowd out all the useful information in the cache.
- Instead of using c[r] to index into the cache, you do c[r + secretKey].
- Weak references vs strong references in associative arrays. The former says that when the system feels like it, it can discard items from the array/cache. A strong reference is one that is permanent (not gonna go away until the program explicitly asks for). The weak reference method is good because we likely don't need all of the items.
- Storing arrays is hard. You can try static allocation in that you look at the program before it runs, and you basically map out where everything will be placed into memory.
    - The pro with static allocation is that you don't have any overhead when the program is actually running. Another is that you won't have any crashes due to storage exhaustion. You'll know if you don't have enough memory before you even start running. Another is that it will let the compiler do some optimizations.
    - A downside with static allocation is that you cannot really change sizes and you could end up wasting a lot of space because you over allocate. You also can't really have any temporary arrays because they are useful. It's also hard to determine how much to allocate if you have conditional statements.
- Guard pages are used to make sure that there isn't one recursive function that is allocating a shit ton of memory. Once the program tries to allocate memory in that area, it will crash.
    - Need to make sure that the guard page is large enough so that the memory allocation doesn't just jump over the guard page.
- For nested functions, you need to have both the definer's bp as well as the caller's bp.

- Keeping track of the roots in heaps
    - In C++/C, programmers can call free and del which is hella fast but you may have dangling pointers or you may free too late and have a memory leak.

2/23 - Week 7

- The dot represents that there is an individual element, usually comes from cons.
    - The last element is a cons that has both of the elements inside of the pair.
- Racket is an implementation of Scheme, and Scheme itself is really just an API.

2/27 - Week 8

- Every value in Scheme is an object. In Java, you have ints, and floats, and whatever, so in some sense Scheme is object oriented. But you don't have classes, and the objects are allocated dynamically, are never freed, and assumes a garbage collector.
- Types are latent (the objects' types are determined dynamically. You cannot just look at it and determine what the types are) and not manifest (type where you can look at the program and determine all the types, so basically static type checking).
- Scheme uses static scoping, which means we can determine scopes just by looking at the program. Every language basically does that, but dynamic scoping is used in Lisp.
    - To figure out scope in Lisp, you look in the current function, then look in the caller, then in the caller's caller, etc.
    - Static scoping means that you look in the current function, then in the definer, and then in definer's definer.
- Dynamic scope is cheaper since you don't need the static chain. However, with static scoping, you have more reliability since you can just look at a program and you'll know how it should work. It is also has more efficiency since the compiler and the hardware will know what to do. Thus, static scoping is used now.
- Scheme is call by value. Prolog isn't call by value though.
- Wide variety of object types, which include procedures. They are just objects and we can pass them around just like anything else.
- Continuations are specific procedures.
- Easy to write Scheme programs that generate other Scheme programs.
- Eval will run through a piece of data and give you the output.
- Tail recursive optimization is required.
    - If the last thing f does is call g, and then return whatever g returns, then don't grow the stack.
    - Basically after function f (the one you call in your recursive case) returns, there shouldn't be any other operation that occurs.
        - Such as the add operation in fib
- No integer overflow in Scheme. Basically a special case of memory exhaustion.
- Identifiers in Scheme are a-zA-Z0-9+-.?*/⇔:$%^&_~@
    - A lot of these are part of the syntax in other languages, but in Scheme all of these are identifiers.

- 0-9+-. Cannot start identifiers because we want to distinguish identifiers from numbers.
- +, -, …, ->, * are something
- These identifiers are case insensitive, and are used are variables, symbols, and/or keywords.
- #t and #f are boolean values.
    - #f is the only false value. Even 0 is evaluated as true.
- Lists and vectors are different. Lists are built out of pairs and it's easy to build sublists from it. Vectors are more like blobs that we can index into. It's more compact but less flexible.
- String is technically a vector of characters.
    - #\c stands for the character c.
- The quote means that we want to yield a symbol without evaluating it.
- Quasiquote allows you to start quoting at the beginning and then stop quoting and start evaluating.
    - ‘(+, x y)
    - The comma acts as the antiquote.
- Scheme keywords
    - (define id E) where id is an identifier and E is the expression we want it to equal.
- (lambda (x y) E) means you're going to evaluate E later. It's like define I guess.
- (lambda x E) means that x will be bound to a list of values.
- And as well as Or do short circuit evaluation.
    - They return arbitrary expressions rather than booleans.
        - (or (“First Check”) (“Backup”)) shows how this will return the first check if it's true, and then backup if the first one ended up being false.
- And, or, cond are all syntactic sugar variations of if.

3/1 - Week 8

- Namedlet gives you convenient recursion.
    - (define (reverse l) (if (null? l) ‘() (append (reverse (cdr l)) (list (car l)))))
    - This is O(n^2) because append makes clones of the arguments (since it doesn't want to modify them)
    - It allows you to define a function and immediately call it.
- The more complicated version would be the one where you have an updated list argument. That argument acts as an accumulator that we return at the end.
    - The problem is that we've exposed two functions to the world since we need to have a temporary function that takes in an empty list to start with. Instead of creating that separate function, we want it to be part of the implementation itself.
- The best version
    - (define (reverse l) (let revHelper ((l l) (a ‘())) (if (null? l) a (revHelper (cdr l) (cons (car l) a)))))
    - This is a tail recursive function and it sort of like a while loop. We know because the if is in the last position, and thus the two control jumps are at the end as well.

- Identifiers can be symbols (those with single quotes before them), variables, or keywords (lambda, if, define, etc)
- (define-syntax and (syntax-rules () ((and) #t) ((and x) x) ((and x1 x2 ...) (something)) .))
    - Pattern matching at compile time, not runtime.
    - And isn't really built into the language, it's something that we define.
    - Cannot do this in Java or C++, but we can define operations like that here.
- (define-syntax or (syntax-rules () ((or) #f) ((or x) x) ((or x1 x2 ...) (something)) ….))
    - But apparently that's not right, so this is the real version
    - (define-syntax or (syntax-rules () ((or) #f) ((or x) x) ((or x1 x2 ...) (let (t x1)) (if t t (or x2 ...))) ….))
- Problem of capture is when a macro has the same name as the argument to the function you're interested in.
    - The fix in Scheme is to do scope first before macro expansion.
- Scheme primitives
    - (cons x y) builds a pair containing x and y. Car and cdr access those values.
- (Eq? x y) is pointer comparison (so it's very fast, just one instruction), while (= x y) refers to number equality and thus the arguments have to be numbers while (eqv? x y) is like dereferencing the objects, and then comparing. (equal? x y) is another one and it will look recursively at all the objects inside of x and y. This would be great for comparing the contents of nested lists. The middle two are O(N) and the last one is pretty expensive, it even might not return in some cases.
    - There are some objects that are not eq? But they are =.
        - (= 2 2.0) is true while (Eq? 2 2.0) is false
    - The converse will be true with NaN and infinity stuff
- Cool to know that car L does not do any cloning, it simply gives you back the value.
- The answer to (Eq? 2.0 2.0) is that it depends on the implementation.
- Comparison and identify and language is important in Scheme.
- The Scheme interpreter must keep track of two things: the instruction to be executed next (instruction pointer), and the environment in which that next instruction will be executed (base/framer pointer). This can be represented as a pair, and stored in a register, and the values in that pair will get updated regularly.
- Continuations allow you to take a snapshot at the values of that pair at runtime and you can figure out what the interpreter will do next.
- In a way, the interpreter is maintaining a list of things to do.
    - We can use continuations to change the order of that to do list.
- A continuation is a small object that contains an instruction pointer and an environment pointer. To create a continuation, ask the interpreter what its current values are. You cannot look at its values but you have the object.
- The function that gets you the object is called (call/cc p). This function creates a continuation object k, and then calls p (the function that you passed in) with the argument k, and then return whatever step 2 returns.
    - P is normally chosen such that you can copy the created continuation k over to a local variable or something.

- To use a continuation k, you pass it a value v, and then the interpreter sets its own ip and ep values to that of k. This is done by (k v)
    - Tells interpreter to stop doing what you're doing, and to go back to those ip and ep values. Then, call/cc returns v.
    - This (k v) does not return, but the call/cc is what returns once or multiple times.
- Continuations are read only, and to use it, you call it with (k v)
- Using this continuation stuff

```
(define (prod ls)
    (call/cc
        (lambda (k)
            (let prodf ((ls ls))
                (cond ((null? ls) 1)
                      ((zero? (car ls)) (k 0))
                      (else (* (car ls) (prodf (cdr ls))))))))
)
```

- This code will make sure that if a zero is reached, then nothing will occur and no multiplications will have happened.
- Threads with continuation

```
(define tlist '())
(define (thr thunk)
    (set tlist (append tlist (list thunk))))

(define (start)
    (let ((next (car tlist))
        set tlist (cdr tlist))
        (next))
)

(define (yield)
    (call/cc (lambda (k)
        (thr k)
        (start))))
```

- Side note: To use a continuation, you must use an argument.

3/2 - Week 8

- Want all functions to be non-blocking since you want the server to respond to all requests.
- You'll know the connections between the servers but not those between the clients.
- Every server should accept TCP connections from clients using telnet.
- In IAMAT command, client tells the server this is my name (arg 1), the longitude/latitude (arg 2), and then the time (arg 3).

- Server maintains a list of clients that it has received something from.
- In WHATSAT, you'll give the client name/address, then the radius, and then the number of results you get back.
    - This is where you do the calling to Google Places API. The client doesn't do this but it tells the server which will send it.
- When a server gets a message, it needs to check its table to see if it has info on that client, and if it doesn't, then it needs to talk with another server that is connected to in order to ask for more information and see if they have info in their table.
- Servers need to communicate their tables to the other servers and server operation should continue even if another one goes down.

3/6 - Week 9

- There is a style of writing continuations even if you're using languages that don't have them built-in.
    - The basic idea is that we pass an extra argument to each function that specifies what it should do after it finishes. In effect, it tells us where the function should return to.
- Heaps are storages that aren't LIFO stacks. You can allocate and free objects in whatever manner you like.
- Interpreter needs to record every pointer that can point into the heap. Those roots could be on stack, in global vars, or in registers.
- malloc(100) has to find a region in the heap where you can set aside 100 contiguous bytes of memory and then return a pointer to that memory.
- You can keep track of all the free areas in the system by having a free list data structure that knows the size and location of the free spaces.
- Fragmentation occurs when you have a bunch of free small areas, such that you cannot allocate the object, but you would have been able to if the individual blocks were combined together.
- Early entries on the free list tend to be really small because you keep chipping away at those.
    - This affects the speed since you're looking through all the small entries, and getting rejected, and this will slow down malloc.
    - Roving pointer says that we set the free list pointer will point at the place that got allocated last so it's not necessarily pointed to the beginning.
- When you free memory, you want to also coalesce freed blocks. How do we figure out what's to the left and right? A naive solution is just walking through the whole free list and determine what's to the left and right. This is a O(N) operation. Instead, we have metadata associated with each block to say whether the previous block is taken/free and if free, how much space you have. Trading space for CPU time.
- Realloc gives you the chance to modify how much memory that object takes up. To shrink an object, you free its tail. To grow an object, you can shrink the side of the adjacent free block if that space is indeed free. But, if you want to grow an object and it

is next to a free area that isn't big enough, then you have to treat this as an malloc + copy + free.
- Moving blocks within the memory is not really done in C and C++.
- We carve out some space at the beginning of each free block and store information from the free list. Those boxes will point to one another.
- Mmap is a function that changes the current process's page table.
- For larger objects, malloc can use mmap, but this will be more expensive than necessary for smaller objects.
- Garbage collection solves problems where 1) you try to use pointers when you freed them or 2) you forget to free memory.
    - 1) is dangling pointers and 2) is memory leaks. 2 is a performance issue while 1 is a correctness issue.
- Idea for garbage collection is that we tell programmers that they don't need to use the free function. It may free objects at some later time, but it will eventually.
- Garbage collectors use "mark and sweep" which means that it will look at the roots that point into objects in the heap, and we'll find objects in the heap that are not pointed to by any of the roots or any of the other objects. Basically, we're looking to find unreachable objects.
    - Mark is when we find all the reachable objects, and mark them.
    - Sweep is when we free all unmarked objects.
- The downside is that the above two steps are relatively expensive. Program, as a result, will freeze occasionally, which isn't ideal for programs that require good latency all the time.
- You can create a GC for C, C++ by using an external garbage collector instead of relying on free.
    - We use a conservative GC, because in C,C++ we don't know the roots. The object layout is not known.
    - Works by scanning through the stack (all of static memory), and looking at whether or not each value looks like it's a pointer. The conservative GC can make a lot of mistakes.

3/8 - Week 9

- For Java garbage collection, you have a couple of approaches.
- Generation based collection is where you divide your object into older and newer ones. Allocation of storage in the nursery (where the new ones are) will be inexpensive.
- There is also a copying collector which means it is allowed to move objects around as the garbage collector, but when you move an object, you have to update all the pointers to those objects.
    - Let's say you want to garbage collect the nursery. We'll figure out what roots point into the nursery and mark the ones that can be trashed. Then, we allocate a new nursery and update all the roots to point to this new nursery location instead.

The new nursery location will now contain adjacent blocks. We've "squeezed out" the garbage. We also modify where the heap pointer will point to.

- One pro is that our work is proportional to the number of objects used, not the total objects.
- Another is that the cache lines are used more efficiently in that in use objects are packed into the same cache because they are adjacent and we don't need to touch the free objects. This is kinda like mark and sweep. All the work you're doing doesn't need to look at the free area.
- In multithreaded garbage collection, each thread has its own nursery and thus each thread has its own heap pointer, and thus the malloc operation will be very fast.
- Real-time garbage collection is for those applications that need to be fast with garbage collection because of the latency requirements.
- One of the main ideas with real-time GC is that you need to put a hard upper bound on the time it takes for new(). A full fledged garbage collector with new() will take too much time. Therefore, we want to be able to do incremental garbage collection.
- Every time you call new, system will allocate storage for you, and then does a couple of instructions (only a couple because of the hard upper bound) that looks for garbage of collect. If you make enough new() calls, then you'll eventually garbage collect everything.
- One of the issues is that your garbage collector will be running while your own program is running, which could create issues where the G.C will mutate something that your program might be unaware of.
    - G.C in a different thread is one possible solution to this, but it's tricky because your object space is changing.
- Garbage collection in Python is very implementation dependent. G.C in Python uses reference counts. Every object in Python will have a little piece of info describing information about it, such as a count of the number of pointers to that object.
    - P = q will increment the reference count of q and decrements the reference count of the thing that p used to point to.
- Every time assignment statement in Python is not just a load/store but also increments/decrements of reference counts. This slows down assignment but when the reference count drops to 0, you know that you can reclaim the storage immediately and free the object. You don't need a fancy garbage collector.
- The catch is reference cycles which refers to the problem where two dictionaries p and q, and each has an entry that points to the other dictionary.
    - P[i] = q and q[j] = p and q = 42 and p = 42
    - Adding q[j] = 42 as the middle instruction will break the cycle
- There's no real solution, and this causes memory leaks, so we just ask programmers not to do this stuff.
- Recent versions of Python do these reference counts, but when the memory gets low, then the conventional mark and sweep is used.
- Explicit nulling is setting some objects to NULL in attempts to hint to the garbage collector that this value is garbage. But now, we don't use this that often because it's lowkey confusing.

- You can also have free lists that represent quick places where you can get some free memory.
- Python is similar but also different from the other languages.
- Python was based on FORTRAN, but a problem was that the language was only for experts. BASIC was the easier and more stripped down version of FORTRAN.
- Python is a reaction against Perl and also has some of the best components of ABC. The grammar syntax came from trying to teach in the best way.
- In Python, the indentation requirements make it so that you cannot represent Python grammar with BNF.
- Python is object oriented, and thus every value is an object. This is like Scheme and OCaml, but is not like Java and C++.
- Each object has an identify, type, and value. The first two cannot be changed, but the 3rd can be changed if the type is mutable.
- “Is” is a built in operation and is true if a and b have the same identify.
- “==” is also a built in operation and is true if a and b have the same values.
- Functions are also considered as objects.
- Python has multiple inheritance.
- An import statement first creates a namespace, then reads and executes a source file in this namespace. Then, it creates a new name “moduleName” in the caller’s namespace so that you can say moduleName.function().
- Python built in types are numbers, sequences, mappings, and callables.
    - Sequences can be broken up into string, Buffer, List, Tuple, etc.
        - Operations include access using bracket notation

3/9 - Week 9

- Each application is running Tensorflow and it has a Python top level, and that creates a model file that gets fed into C++ layer.
- The C++/CUDA part is the most computationally expensive part because that is where the training and inference happens.
- For Hw6, found out that the Python part was the bottleneck, so we want to explore alternative languages and they are OCaml, Java, or Julia/Haskell/Rust
- Highlight key differences between the languages. Highlight the relevant language details. Go into quick overview of how you would implement the project with each of the languages, this would involve going over the language features and why use that language feature
- Strongly typed languages force you to make conversions very explicit, instead of implicit.
- Dynamically typed means that type checking happens at runtime, not compile time
- With generators, they are functions where you can stop and resume their executions.
- When you call a generator function, a generator object that stores the function code is created instead of the function actually being run.
    - Calling next on gen will run the function until yield is hit, and that current value is returned.

    - Every time you call next, the execution gets resumed until the yield.
- Yield from f(x) says that we delegate the yields to f(x). The return value of f(x) is the result of the yield from expression.
- All generators are coroutines but not all coroutines are generators. Coroutines use yield to communicate with their caller.
- With coroutines, we allow caller to send info back to generator in the middle of the generator's execution.

3/13 - Week 10

- Object oriented languages are C++, Java, OCaml, Python, and Scheme
    - The first 3 use static checking (do the checking at compile time for safety), and the latter 2 have dynamic checking (do it at runtime for flexibility and performance)
- An object oriented language isn't the same as object oriented programming (or style)
- Class based vs prototype based languages
    - In class based, you try to specify the types of objects. Two ways of creating objects (new and clone)
    - In prototype based, there are no classes, there are only objects. You can only clone objects. Instead of a class C, you have a prototype object P. Every object can have a different set of fields and methods (contrast to a class where each object of the class needs to have certain fields/methods), but in practice we use clone a lot, so most of the times we have the same fields, but you have the flexibility to change this.
- In prototype based languages, you get more flexibility and is simpler, but there is a downside in performance. However, we can do JIT compilation and that would help a prototype based language more than it would help class based ones.
- Single vs multiple inheritance
    - We need to determine if a subclass can delete parent methods, determine what we inherit (methods, variables, etc)
    - Subtyping vs subclassing which refers to private inheritance in C++
- Parameter passing can either take the form of call by value or by reference
    - In by value, you have f(E1, E2) the caller evaluates the expressions, stores the values in a copy, and passes those copies to the callee.
        - This is the most common approach because it's simple but this is slow with large values as arguments.
    - In by reference, we compute the address of the arguments, but doesn't compute the value. This is much more efficient for large values and gives a way for callee to communicate info back to the caller, but this is slower for small objects and it's dangerous af to have your callees be able to modify your arguments.
- Call by reference runs into problems when the compiler tries to optimize your code because of aliasing. For example, when you have 2 arguments, and you pass the

argument to both locations. This mean that compiler has to stop optimizing since it would change the behaviour of the code.
    - If you know that the two arguments have to be distinct, then the compiler can indeed make that optimization.
    - You can do that through int func(int *restrict px, int *restrict py){ … } where the keywords make sure that px and py don't alias each other.
- Call by result is like x = p() whereas the syntax is actually p(x). In the callee, x is uninitialized, some computation is performed, and x is returned.
- Call by macro expansion means that you're not doing computation at runtime, and you're looking for keyword functions you already have the compilation equivalent.
    - Downside is that scope is tricky, and the recursion is limited (a macro that gets calling itself at runtime is a little messy)
    - Upside is a little performance and flexibility in that you can change them as necessary.
- Call by name means that we use procedures in the same way that call by reference uses pointers. You have pointers to procedures and then you call that procedure? Doesn't actually evaluate anything, but just create a function such that if you call it, then it will evaluate the result. Helpful when you don't want to evaluate the arguments, but you want the arguments to be a thunk and reevaluate the function for multiple values. Basically, this is cool for when we want to pass in a sub function that we will use in the callee to help get the final result.
    - Call by name is also safer than call by value in that because you don't really evaluate your arguments, you can avoid some divide by zero errors or something. You can also avoid infinite loops. Call by value is termed eager evaluation, you evaluate all the arguments even though you may not need all of them. Lazy evaluation is when you put off evaluating the arguments until later. Downside to lazy eval revolves around performance issues.
- Call by need is like call by name except it only calls the thunks once. It memoizes in a certain way, so that it only runs the computation once, saves it, and returns the saved answer after that.
- One of the main parts of programming languages is handling errors. An error can be thought of as the logic problem in your head. A fault is a latent problem in program (problem with the program, not the execution). A failure is a bad execution with observable symptoms.
    - Exception handling catches failures, not faults.
- Compile time checks are often the best way to handle faults. Most reliable way to prevent program from failing.
    - Static type checking is an example of this. They're great but they are less flexible.
- You can also try to handle this through preconditions where you tell the callers that this method will be undefined if you don't make sure that the following conditions are met.
- Fatal errors are those that cause your problem to crash and dump core. The worse situation is when the error leads to undefined behaviour. The best way is to try to catch

these errors using try catch exceptions. We need to have a runtime data structure that tells the program where to throw the exception to and what to do after that.

3/15 - Week 10

- Cost model shows the difference between absolute vs O() costs. The model is your model of how much programs with cost and resources. Coming up with a good cost model involves thinking about **what costs we are worried about** because there are a number of different things.
    - CPU Time - number of CPU cycles and include summation of time for each additional processor
    - real time - time includes I/O devices and context switching delays
    - I/O costs - might be beneficial to eat up more CPU time in order to reduce I/O operations
    - RAM (or memory) and cache and disk space
    - Energy consumption - Normally correlated with CPU time, but you can run CPU in a slow mode where less power is used
    - Network usage - Two features are called throughput (number of bytes per second) and latency (time taken for one round trip)
- Throughout is associated with CPU time, but the overall latency is associated with real time.
- Example cost model for a lisp list - linked list of pairs
    - Prepending is cost b/c it's a linked list, and you just call cons. Didn't have to check how long l is, or allocate more storage, or copy values to new list. People thus write code based on the assumption that cons will be cheap. The cost is O(1).
    - Appending, however, is expensive because we make a copy of the list, and when we get to the end of the copy, we add the new value. The cost is O(N). Append can also take variable number of arguments. Doesn't need to make a copy of the last argument, but do need to make a copy of the first n-1 lists. We use the copy approach instead of modifying the arguments since we want programmers to code functionally.
    - Length is an O(N) operation.
    - Null? Just needs to check if the first arg is empty. This is O(1) operation. Thus, you would never ever check (> (length l) 0) because that's hella expensive.
- Prolog cost model
    - For some unification X = Y, if X is just some variable, then the cost is O(1) because we just have to store a pointer for Y, and same vice versa. When X and Y are both non-variables and thus are complicated structures then the interpreter walks through X and walks through Y and looks for mismatches. The worst case is proportional to the size of X and Y. O(min (|X|, |Y|)).
    - Unify with occurs check(X,Y) will be O(max (|X|, |Y|)).
- Scheme cost model

    - (eq? A B) is just O(1) since it looks at pointers. But eqv is O(n) and equal could loop forever. This is a main reason for why we have these different versions of checking equality.
- Procedure calls in C, C++, and Java
    - Since our programs do so many of these calls, we should know the costs associated with them.
    - In a call, you have to evaluate the arguments, copy argument values to parameters, jump to procedure start. That's all done by the caller. The callee needs to allocate a frame and save registers, and finally do the computation. Then, it needs to copy the results to the return register, deallocate the frame, and then go to the return address. 90% of this stuff is all paperwork overhead not really associated with the computation.
    - To make this faster, we can use tail recursion optimization which allows you to share frames and registers, which thus saves memory.
    - Inlining is another option where you take all the code in g and copy it into f which means that you don't need to deal with new frames or return address, etc. The reason we don't inline everything is that this increases program size which has to be stored somewhere in memory. Your program will consume more space in your instruction cache. Also, it kills recursion. Also, there is a very tight binding between the caller and callee which is not ideal, and you have to know the source code of the callee which isn't available sometimes.
- Array access costs
    - The equivalent instruction is &a + (i - &b(a)) * sizeof a[0]
    - This gets hella expensive, so that's why we try to make sizeof always equal power of 2 so that we can have shifts instead of multiplies.
- Something about static semantics, dynamic semantics, and attribute grammar.
- Denotational semantics for functional
- In strongly vs weakly typed, in a weakly typed language, the type of a value depends on how it is used and in a strongly typed one, a value has a specific type and that type cannot change.

3/16 - Week 10

- Function in OCaml is something that takes in an argument and pattern matches with it.

Textbook Notes

Chapter 1 - Programming Languages

- An imperative language contains two hallmarks: assignment and iteration
    - C is an example of an imperative language.
- Functional programs have the hallmarks of recursion and single valued variables.
    - ML is an example of a functional language.

- Expressing a program in terms of rules about logical inference is the hallmark of logic programming.
    - Not great for computing mathematical functions.
    - Prolog is an example of a logic language.
- Java is an object oriented language, which means it is imperative and it uses objects, which are bundles of data that know how to do things to itself.
    - Helps keep large programs organized.
- Each language favors a particular programming style, which is a particular approach to algorithmic problem solving.
- Large standardized libraries of predefined code are called APIs.

Chapter 2 - Defining Program Syntax

- The syntax of a language is the part of the definition that says how programs look.
- Semantics of a language are the parts of the definition that says what programs do.
- The 4 parts of a grammar are the set of tokens, the set of non-terminal symbols, the set of productions, and a non-terminal symbol called the start symbol.
    - Tokens are the smallest units of syntax: Keywords like idf or operators or key words like Fred.
    - Non-terminal symbols are different language constructs: A noun phrase or a verb or a statement or expression.
    - Production consists of a LHS and RHS where the left has a single non-terminal symbol and the right has a sequence which can either be a token or a non-terminal symbol.
    - Start symbol: Non-terminal symbol
- Language can have phrase structure and the lexical structure.
    - Phrase structure shows how to construct parse trees with tokens at the leaves.
    - Lexical structure shows how to divide the program text into tokens.
    - Both structures can be specified by two different grammars.
- A scanner will read an input file and convert it into tokens whole a arser will read the tokens are form a parse tree.
- Languages that abandoned the column-oriented approach are free format languages.
- Difference between BNF and EBNF is that EBNF will have a few more metasymbols which are part of the language of the definition, not part of the language being defined.
    - [something] in the RHS means that something is optional.
    - { something } in the RHS means that something can be repeated any number of times.
    - Any language that extends BNF in this way is called an EBNF.
- Syntax diagrams use a directional graph to show the productions for each non-terminal symbol.
- Formal languages contain context free grammars which mean that the children of a node in the parse tree depend only on that node's non-terminal symbol, not on the context of the neighboring nodes in the tree.

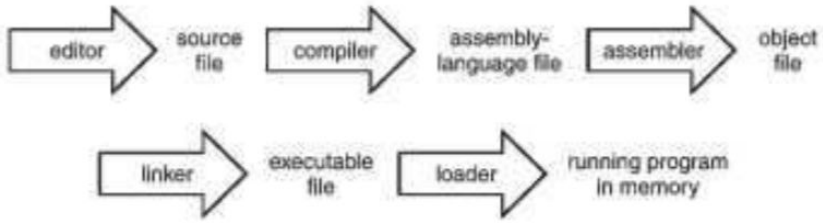
    - Regular grammars are less expressive and context-sensitive grammars are more expressive.
- Grammars are used to define the syntax of programming languages, both at the level of the lexical structure which is the division into meaningful tokens and at the phrase structure where we have the organization of tokens into a parse tree showing how the meaningful structures of a program are organized.

Chapter 3 - Where Syntax Meets Semantics

- A grammar is a set of rules for constructing parse trees.
    - Language defined by the grammar is the set of fringes of the parse trees constructed according to those rules.
- Grammars must generate unique parse trees that correspond to the desired semantics for the language.
- Operators refer to the tokens used to specify operations (+, *, -, etc) and to the operations themselves.
    - Inputs to an operator are called its operands.
- Unary operator takes a single operand, binary takes two, ternary takes three.
- Infix notation is when the operator is written in between the two operands.
- When an expression contains a sequence of operations that aren't parenthesized and are all at the same level, then the operator is left associative.

Chapter 4 - Language Systems

- An integrated development environment (IDE) is a language system that provides the programmer a single interface for editing, running, and debugging programs.
    - Unintegrated system is where those components are separated.
- In those types of systems, the programmer creates a text file or source file that contains the program. This, itself, cannot be run by the hardware. It has to be processed by a compiler, whose job it is to translate programs into a lower level language, like assembly language. Then, an assembler processes the assembly code and converts each instruction into machine language which is binary. The result is an object file that isn't really readable. But this still isn't ready because the assembler still doesn't know where to place variables in memory or where references are to functions, etc. The linker then collects and combines different parts of the program, so this is where it will search for function calls and add those pieces of code to the program. The linker's output is a single executable file. Then the system loader gets the program into memory and now you can finally run it.
- Different blocks for methods can be loaded in at different memory locations. Main can be at 20, fred can be at 60, etc.
    - Each piece is loaded somewhere and all references to that piece are resolved at this point.



- You can do the above steps all at once by just doing gcc main.c or you can set certain flags so that you do them one at a time.
- Code generated by a compiler usually optimized to make it run faster and/or take less memory.
- Loop invariant removal is where you can take some unchanging computation that is originally in your while loop, and remove it outside so that you only have to do it 1 time instead of having to do it n times.
- Some compilers will reorganize the code, some add specialized copies of the code in different places, some remove unnecessary pieces of code.
- Carry out the steps specified by a program, without first translating it to another language is called an interpreter for the language.
    - Language systems that interpret a program run it slower than those that compile the program.
    - Compiling takes more time up front, but then the program can run at hardware speed.
- You can also create executable code that doesn't necessarily run on hardware, but runs on a virtual machine, which would require a software simulator, which would carry out the steps indicated by the virtual machine language program.
    - The interpreter for this type of intermediate code can be implemented on many different machines.
- Java systems compile to an intermediate code, known as the Java virtual machine.
    - Browser can run different applets by interpreting the bytecode file, which is a file in the machine language format of the JVM.
- Interpreted systems need to tokenize a program, which means converting it into a sequence of tokens where all the keywords and the punctuation are identified.
- A real machine is implemented in hardware while a virtual machine is implemented in software.
- One thing that can also happen is delaying the linking step, so that the code for library functions is not included in the executable. Instead, a loader will find the .dll files required and links a program to all the library functions it needs just before the program starts to run.
    - Above is load time dynamic linking.
- Run time dynamic linking is used less often, but in this one, the program must make explicit calls to find the .dll file at runtime and that's when it loads functions within that library.

- There are some libraries of functions for delayed linking that are stored in files that end in .so and they are called shard libraries and are linked to a program by the loader just before the program starts running.
    - You can also call these files dynamically loaded libraries where the program must make explicit calls to find the library during runtime.
- The class loader has to verify the bytecode for each of the loaded classes.
- The pros with using delayed linking is that multiple c=programs can share a copy of the library function, the function can be updated independently of the program, and you can sometimes avoid loading functions that are never called.
- Compiler profiling is when the compiler makes guesses about which parts of the program will be executed most often, and thus which parts of the code need to be optimized the most.
- Profiling includes compiling the program twice. After the first time, the program is linked, loaded, run, and profiled. The last step collects stats about the runtime behaviour of the program, how many times each piece of the program is executed, and on the second pass, the compiler uses that info to generate better code.
- Dynamic compilation is where some compiling takes place after the program starts running.
    - Simple version is only compiling something when it is actually called.
    - Can be called Just in Time compilation
- The act of associating properties (domain, type, memory location) with names is called binding.
- Language definition time is when properties are bound when the language itself was defined or created. For example, the keywords void and for are part of the language definition.
- Language implementation time is when properties are determined based on the implementation of the language. For example, the range of int values for some machines may be different.
- Some properties are determined at compile time. For example, the type of variable i.
- Link time is when the linker finds the definitions of library functions to match each of the references in the program.
- Load time is when the loader puts the finishing touches on a program before it runs. For example, the memory addresses for variables get bound at this time.
- Runtime is when values of variable i get bound.
- Early binding refers to all times before runtime, and late binding is basically runtime.
- In a pure interpreter, there is no compiler time.
- Runtime support refers to concepts that need to be handled at runtime. Some examples are startup processing, exception handling, memory management, and concurrent execution.

Chapter 5 - A First Look at ML

- ML is a functional language.

- ML maintains a variable named it whose value is always the value of the last expression that was typed in.
- Numeric constants can either be ints or reals.
    - 1234 is an int while 123.4 is a real
- Negation operation is performed with a ~
- The constants true and false are the two values for ML's bool type.
- String constants are enclosed in double quotes.
    - To get a character constant, put a # before the double quote.
- ML supports escape sequences like \t for a tab, \n for a linefeed, and \" to put a quote without ending the string.
- ^ is the concatenation operator for strings.
- Logical or is **orelse**, logical and is **andalso**, and logical complement is **not**.
- Integer division is expressed used div instead of /.
- ML has an if <exp> then <exp> else <exp> format for conditionals.
- When the same operator works differently on different types of operands, then it is said to be overloaded.
- You can multiply two reals together or two ints together, but not one int and one real.
- ML does have some conversion functions like real(), floor(), round(), trunc(), etc.
- To call a function in ML, you just write the function's name followed by the parameter.
- Val keyword is used to define a new variable and bind it to a value.
- Garbage collection is when the system is reclaiming pieces of memory that are no longer used.
- Given two ML types a and b, a*b is the ML type for tuples of two things.
- To extract the ith element of a tuple v, write the expression #i v.
- There cannot be a tuple of size 1 in ML.
- All elements of a list in ML must be of the same type.
- Names that begin with an apostrophe are called type variables, which stand for a type that is unknown.
- The @ operator is used to concatenate two lists. Both arguments have to be lists.
- To define a new function in ML, you give a fun definition.
    - fun <function-name> <parameter> = <expression>;
- The -> symbol is a type constructor where given types a and b, a -> b is the type for functions that take in parameter of type a and return a result of type b.
- Polymorphic functions allow parameters of different types.

Chapter 6 - Types

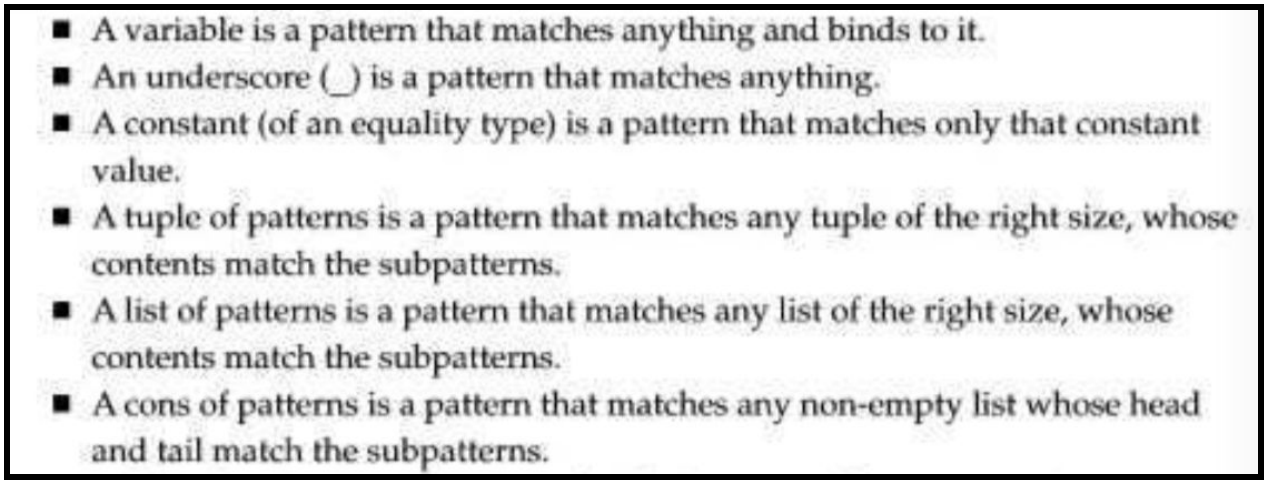
- A type is a set. When you declare that a variable is of a certain type, you are saying that the values that the variable can have are of a certain set.
    - All elements of that set share a common representation in that they are encoded in the memory in the same way.
- A type that a program can use but not define for itself is a primitive type in the language, but a type that a program can define for itself is a constructed type.

- The definition of each programming language describes the primitive types for that language.
- Some languages (ML and C) are languages that allow different implementations to define primitive types differently.
- Programmers can define new types through enumeration. This way you can just list all the elements of that type.
    - Type primaryColors = {red, blue, green}
- A subtype is a subset of the values, but it can support a superset of the operations.
- For any two sets A, B the notation A -> B refers to the set of all functions with inputs from set A and outputs from the set B.
- With a type annotation, the programmer supplies explicit type information to the language system. They make the program easy to read, and the language system could use this information for optimizations.
- Static type checking determines a type for everything before running a program. It gets this info through type annotations and through type inferences which it does by themselves.
- Dynamic type check is the same test performed at runtime. The system will run the program without predicting whether there will be type errors.
    - Hard to execute as efficiently since we need to find out the types and keep track of them as the program is running.
- One unique case is where the compiler might know the static type of a formal parameter to a function but then during runtime, the actual parameter might belong to a subtype.
- Purpose of type checking is to prevent the application of operations to incorrect types of operands.
    - Strongly typed languages make sure that no applications are type incorrect.
- The strength of type checking is inversely proportional to the number of holes they have.
- Name equivalence is the type equivalence rule that says that two types are equivalent if and only if they have the same name.
- Structural equivalence says that two types are equivalent if and only if they are constructed from the same primitive types using the same constructors in the same order.

Chapter 7 - A Second Look at ML

- ML automatically tries to match values with its patterns and takes action depending on whether or not they match.
    - n matches any parameter while (a,b) only matches the arguments which are tuples. We've also created two variables a and b with that last one.
- The underscore character is a pattern that matches anything and doesn't introduce any new variables.
- Fun f _ = "yes" is a function that can be applied to any parameter of any type and the result will always be a string called "yes".

- Fun f 0 = “yes” is only defined with you call f with the argument 0. Also, we will get a warning saying that our match is non exhaustive because the function does not handle cases where the passed in argument is not 0.
    - If you call f 1, then you’ll get a runtime error.



- Summary above
- You cannot use a pattern with the same variable name more than once.
    - Fun f (a, a)
- Variables defined in a let expression between let and in are visible only from the point of definition to end.
    - Let expression is used for local function definitions.
- Function definitions can be made inside another function definitions.

Chapter 8 - Polymorphism

- Functions with extra flexibility, meaning they they can accept multiple types, are called polymorphic.
- Overloading is a kind of polymorphism. An overloaded function name or operator is one that has at least two definition, both of different types.
    - + operator is overloaded since one definition works for integers and another works when the arguments are reals.
    - The system uses the operands’ types to determine which definition of the + operator to apply.
- AN implicit type conversion is called a coercion. Basically, it’s where if you define a variable to be one type and then set it equal to another a value that may not be of that type, the system will force it.
    - Double x; x = 2; will implicitly convert 2 into 2.0.
- These coercions can simplify programmer’s tasks by not requiring them to write obvious type conversions.
- A function shows parametric polymorphism if it has a type that contains one or more type variables.
    - ‘’a * ‘’a -> bool is called a polytype.

- The language system will create separate copies of the polymorphic function, one for each of the different instantiations of the type variables.
- A function shows subtype polymorphism if one or more of its parameter types have subtypes.
- Languages that use dynamic type checking don't have to worry about polymorphism since a function can be called with any type of argument and the suitability is not checked at compile time.

Chapter 9 - A Third Look at ML

- Higher order function is one that takes other functions are parameters or produces them as returned values.

A *rule* is a piece of ML syntax that looks like this:

<*rule*> ::= <*pattern*> => <*expression*>

A *match* consists of one or more rules separated by the | token, like this:

<*match*> ::= <*rule*> | <*rule*> '|' <*match*>

- Rules and matches above
- A rule in a match must have some type of expression on the RHS.
- Case expressions are expressions where the value of the expression is the first rule of the match whose pattern matches the expression.
- It's common in functional languages to have variables that are bound to functions.
- Functions don't have names, but rather we say that functions are currently bound to particular names.
- You can create anonymous functions through fn (a, b) => a > b where you don't have to specify a name. It will automatically get bound to val it.
- A function that does not take any functions are parameters and does not return a function value has order 1.
- A function that takes a function as parameter or returns a function value has order n+1, where n is the order of its highest order parameter or return value.
- Functions of any order greater than one are called higher order functions.
- Currying is when you write a function that takes the first parameter and returns another function, which takes the second parameter and returns the final result.

```
- fun f (a,b) = a + b;
val f = fn : int * int -> int
- fun g a = fn b => a + b;
val g = fn : int -> int -> int
- f (2,3);
val it = 5 : int
- g 2 3;
val it = 5 : int
```

The function g takes the first parameter and returns an anonymous function that takes the second parameter and returns the sum. Notice that f and g are called differently. f expects a tuple, but g expects just the first integer parameter. Because function application in ML is left associative, the expression g 2 3 means (g 2) 3; that is, first apply g to the parameter 2 and then apply the resulting function (the anonymous function g returns) to the parameter 3.

- Currying allows you to create more specialized versions of other functions by calling that other function with particular arguments.
- Map function is used to apply some function to every element of a list. The first parameter is the function and the second is the list.
- Foldr function is used to combine all the elements of a list into one value by taking in a function, a starting value, and a list of values.

```
foldr (fn (a,b) => function body) c x
```

- Foldl does the same but starts at the leftmost element in the list and proceeds from left to right.

Chapter 10 - Scope

- When there are different variables with the same name, there are different possible bindings for that name and each occurrence of the name has to be bound to exactly one variable.
- A definition is anything that establishes a possible binding for a name.
- An occurrence of a name is in the scope of a given definition of that name whenever the definition governs the binding for the occurrence.
- A block is any language construct that contains definitions and also contains the region of the program where those definitions apply.
- Labeled namespace is any language construct that contains definitions and a region of the program where those definitions apply, and also has a name that can be used to access those definitions from outside the construct.
- Scoping rules determined before runtime are static, those decided during are dynamic.

Chapter 11 - A Fourth Look at ML

You can add a parameter to a data constructor by adding the keyword `of` followed by the type of the parameter. Here, for example, is a `datatype` definition for a type `exint` with three data constructors, one of which has a parameter:

```
datatype exint = Value of int | PlusInf | MinusInf;
```

The `exint` type includes three different kinds of things: `Value`, `PlusInf`, and `MinusInf`. By declaring the `Value` data constructor as `Value of int`, we are saying that each `Value` will contain an `int`, which is to be given as a parameter to the data constructor: `Value 3`, `Value 65`, and so on. For each different `int` there is a different possible `Value`. A `Value` is like a wrapper that contains an `int`, while a `PlusInf` or a `MinusInf` is like an empty wrapper. This illustration shows

-
- Info on data constructors above

Chapter 12 - Memory Locations for Variables

- In imperative languages, when a value is assigned to a variable, it must be stored somewhere, which is a memory location associated with the variable.
- In functional languages where you don't really have assignment, you cannot really access memory locations of variables using the & operator.
- Variables are activation specific when their values are dependent on function call and/or the particular control structure.
- Function activations are alive at the same time when one function calls another function before it ends itself.
- All activation specific variables and data like return addresses are put in one block of memory called an activation record.
    - These activation records can be given out statically, meaning one for each function before the program begins running.
- The thing is that there may be cases where more than one activation record is needed for each function. This is the case with recursive calls. That's why we have dynamic stacks of activation records.
- In dynamic, when a function is called, the system prepares to execute it by allocating a new activation record for it, and can be deallocated when the function returns.
- The records form a stack at runtime. They are pushed on call, and popped on return. The activation records are known as stack frames.

Chapter 13 - A First Look at Java

- Primitive types of Java are int, char, double, boolean, void, and null.
- Any value that is not a primitive type is a reference to an object, and the types of such values are called reference types.
    - Class names, interface names, and arrays are reference types.
- When an operator changes something in the environment, the change is a side effect.
- All Java operators are left associative, except for assignment operators.

Chapter 14 - Memory Management

- The OS should grant one or more fixed size regions of memory to use for its dynamic allocation.
- In Java, to create a new array of 100 values, you need to use the keyword new and you need to indicate the size.
    - Int[] a = new int[100];
- An example of unordered runtime memory allocation and deallocation is a heap, which is a pool of blocks of memory, with an interface for unordered runtime memory allocation and deallocation.
- Heap manager cannot combine non-adjacent free blocks.

Chapter 19 - A First Look at Prolog

- A Prolog program is logic, which is a collection of facts and rules for proving things. You don't run the program, you create a "database" of facts and rules which then allow you to pose questions or queries to the program.
- Everything in a Prolog program is built from Prolog terms. There are 3 types: constants, variables, and compound terms.
    - Simplest kind of term is a constant which can be an integer, a real number or an atom.
        - Any name that starts with a lowercase letter is an atom. They may look like variables in any other language, but they are constants in Prolog.
    - Variables are another type and they are any name that begins with an uppercase letter or an underscore.
    - A compound term is an atom followed by a parenthesized, comma separated list of terms.
        - They look like function calls, but they are really just data structures.
- Pattern matching using Prolog terms is called unification. Two terms can unify if there is some way of binding their variables that makes them identical.
- A fact is a term followed by a period.
- An atom that starts a compound term with n parameters is called a predicate of arity n.
- A term is called the head of the rule. It is then followed by the :- token and then by a list of comma separated terms called the conditions of the rule.
- The scope of the definition of a variable is the clause that contains it.
- Prolog is a declarative language where each piece of the program is like a declaration that corresponds to a simple mathematical abstraction, like a formula in first order logic.
- Imperative languages are doomed to subtle side effects and interdependencies.
- The predicate = takes two parameters and is provable if and only if those two parameters can be unified.
- X = 1 + 2 * 3 is shorthand for X = +(1, *(2, 3))
- Prolog lists can contain a mixture of different types of elements.

Chapter 20 - A Second Look at Prolog

- Unification is a substitution or a set of bindings for variables.
- The result of applying a substitution to a term is an instance of that term.
- Term append([1,2], [3,4], [1,2,3,4]) is an instance of the term append([1, 2], [3, 4], Z) using the substitution Z = [1, 2, 3, 4]
- Prolog wants to bind just enough so that the terms unify. We want most general unifiers.
- If you have a variable X and a term t, an MGU will likely be [x -> t], but the exception is if t is a compound term in which the same variable X occurs.
    - This is called an occurs check, whether we check if X occurs in t.
- Most Prolog systems implement unification without the occurs check.
    - If you don't have the check, then you may get an answer like [a | **] which specifies a cyclic term.
- Prolog uses backtracking to explore all possible targets of a procedure call in the order given, until it finds as many successes as the caller requires or until it exhausts all possibilities.
- The Prolog interpreter will use resolution, which is where it applies one clause from a program to make one step of progress on a list of goal terms to be proved.
- Even if a proof tree is infinite, you can still find a proof since the system will search in a DFS left to right fashion.
- Assert(X) adds the term X as a fact in the database. Retract(X) will remove it.
- Cut is a predicate that eliminates backtracking. It won't try to find additional solutions using the clause that actually worked and any subsequent classes as well.

Chapter 22 - A Third Look at Prolog

- The term +(1, *(2, 3)) is different from the constant term 7.
- The predicate is can be used to evaluate terms.
    - X is 1+2*3 and thus X = 7
    - When Prolog tries to solve an is goal, it evaluates the 2nd parameter and unifies the result with the first parameter.
- The second parameter to is cannot have any unbound variables in it.
    - Y is X + 2, X = 1 will cause an error since X is unbound at the time.
- Prolog is dynamically type checked, and thus the system is aware of the type of each value at runtime.
    - Overloading for various combinations of real and integer operands is resolved at runtime, according to the types of the operands.
- In contrast to the is predicate, the = predicate will attempt to unify X and Y in the goal X=Y. This unification doesn't evaluate the numerical expressions but instead pays attention to the term structure.
    - For example, 3 = 2 + 1 would fail.
- Be sure to use =< instead of <=.

Chapter 1 - Introduction

- Scheme has a garbage collector. Atomic values, though, are represented by immediates and don't require allocation or deallocation overhead.

Other Definitions

In computing, **just-in-time** (**JIT**) **compilation**, also known as **dynamic translation**, is a way of executing computer code that involves compilation during execution of a program – at run time – rather than prior to execution.[1] Most often this consists of source code or more commonly bytecode translation to machine code, which is then executed directly. A system implementing a JIT compiler typically continuously analyses the code being executed and identifies parts of the code where the speedup gained from compilation or recompilation would outweigh the overhead of compiling that code.

Java is compiled to an intermediate "byte code" at compilation time. This is in contrast to a language like C that is compiled to machine language at compilation time. The Java byte code cannot be directly executed on hardware the way that compiled C code can. Instead the byte code must be interpreted by the JVM (Java Virtual Machine) at runtime in order to be executed. The primary drawback of a language like C is that when it is compiled, that binary file will only work on one particular architecture (e.g. x86). Interpreted languages like PHP are effectively system independent and rely on a system and architecture specific interpreter. This leads to much greater portability (the same PHP scripts work on Windows machines and Linux machines, etc.). However, this interpretation leads to a significant performance decrease. High-level languages like PHP require more time to interpret than machine-specific instructions that can be executed by the hardware.

Java seeks to find a compromise between a purely compiled language (with no portability) and a purely interpreted language (that is significantly slower). It accomplishes this by compiling the code into a form that is closer to machine language (actually, Java byte code is a machine language, simply for the Java Virtual Machine), but can still be easily transported between architectures. Because Java still requires a software layer for execution (the JVM) it is an interpreted language. However, the interpreter (the JVM) operates on an intermediate form known as byte code rather than on the raw source files. This byte code is generated at compile time by the Java compiler. Therefore, Java is also a compiled language. By operating this way, Java gets some of the benefits of compiled languages, while also getting some of the benefits of interpreted languages. However, it also inherits some limitations from both of these languages.

As Bozho points out, there are some strategies for increasing the performance of Java code (and other byte code languages like .Net) through the use of Just in Time (JIT) compilation. The actual process varies from implementation to implementation based on the requirements, but the end-result is that the original code is compiled into byte code at compile time, but then it is run through a compiler at runtime before it is executed. By doing this, the code can be executed at near-native speeds. Some platforms (I believe .Net does this) saves the result of the JIT

compilation, replacing the byte code. By doing this, all future executions of the program will execute as though the program was natively compiled from the beginning.