The first type of language:
- Programmer: Ada Lovelace (1843)
- High-level: FORTRAN (1954)
- Functional: Lisp (1958)
- Object-oriented: Simula (1962)
- Logic: Prolog (1972)
- Concurrent: Concurrent Pascal (1974)
- Concurrent Actor: PLASMA (1975)
- Scripting: Rexx (1982)
- Multi-paradigm: Oz (1995)

Computation Model:
- Describes a language's semantics.
- A set of coding techniques to solve problems.
- A set of reasoning techniques to prove some properties of a program.

Declarative Computation Model:
- Pure functions (no side effects).
- Stateless (vs. imperative with is stateful.)
- Used in functional, logic, and concurrent programming.

Programming Language: Syntax + Semantics.

Language Syntax:
- Defines what is a legal program vs. what is not.
- Defined by a set of grammar rules.
- In programming languages: sentences `->` statements, words `->` tokens.

Language Semantics:
- Defines what a program does.
- Should be simple, yet expressive.
- Approaches: *Formal Calculus* (lambda, pi, predicate), *Kernel Language* (derived subset), and *Abstract Machine* (Turing machine, details execution).

Lambda Calculus:
- Syntax: `e ::= v | λv.e | (e e)`.
- Semantics: `(λv.E M) => E{M\v}`. This is β-reduction. If expression `M` is applied to function `λv.E`, return `E` where all instances of `v` in `E` are replaced with `M`.
    - *Example*: `(λx.x y)` becomes `x{x/y}` or `y`.
- Currying: Since λ-calculus only supports one parameter per function, you can use composition to simulate it.
    - *Example*: Addition (using currying) would be: `λa.λb.(a + b)`.

- Order of Evaluation: If an expression can be evaluated two ways and terminated, it will have the same result. (Church-Rosser Theorem, called *confluence*.)
    - *Normal Order*: Outer expressions first. Reduce the expression then reduce the result.
    - *Applicative Order*: Inner expressions first. Reduce the arguments then reduce expression.
    - For every expression, if any order terminates, normal order will too.
- Bound & Free Variables: For the expression `λv.e`, all variables `v` in `e` are *bound*. All other variables are *free*.
- α-renaming: To avoid capturing free variables during β-reduction, rename colliding bound variables.
    - *Example*: `(λx.λy.(x y) y)` would become `λy.(y y)` without renaming, which is wrong because `x` (which was unbound to the inner λ) became `y` (which is bound). But if the inner λ had its variable alpha renamed from `y` to `z`, `(λx.λy.(x y) y)` would become `(λx.λz.(x z) y)` or `λz.(y z)`, which is correct.
- Combinator: An expression without free variables.
    - *Recursion (Y)*: `λf.(λx.(f (x x)) λx.(f (x x)))` for normal order or `λf.(λx.(f λy.((x x) y)) λx.(f λy.((x x) y)))` for applicative order.
- η-reduction: `λx.(E x)` can be replaced with `E` as long as `x` is not free in `E` (or bound to the λ we are removing).
- Boolean Logic:
    - *True*: `λx.λy.x`
    - *False*: `λx.λy.y`
    - *If*: `λcond.λthen.λelse.((cond then) else)`
- Church Numerals:
    - 0: `λf.λx.x`
    - 1: `λf.λx.(f x)`
    - 2: `λf.λx.(f (f x))`
    - n: `λf.λx.(f … (f x)…)`
    - Successor: `λn.λf.λx.(f ((n f) x))`
    - Is Zero: `λn.((n λx.false) true)`

Oz Programming Language:

- Variables are immutable.
    - Variable identifier: What you type.
    - Store variable: Part of the memory system.
- Lists: `[1 2 3]` or `1|2|3|nil` or `'|'(1 '|'(2 '|'(3 nil)))`. The `'|'` is cons.
- Top-down Programming: Break a complex task into smaller tasks.
- Iterative Programming: Starts with an initial state, and continually transforms the state until a condition is met.
    - *Schema*:

```
fun {Iterate S IsDone Transform}
     if {IsDone S} then
          S
     else S1 in
          S1 = {Transform S}
          {Iterate S1 IsDone Transform}
     end
end
```

- Constant size execution stack (with tail recursion optimization).
- *Newton's Method (square root)*:

```
fun {Sqrt X}
     Guess = 1.0 in {SqrtIter Guess X}
end

fun {SqrtIter Guess X}
     if {GoodEnough Guess X} then
          Guess
     else
          {SqrtIter {Improve Guess X} X}
     end
end

fun {Improve Guess X}
     (Guess + X/Guess)/2.0
end

fun {GoodEnough Guess X}
     {Abs X - Guess*Guess} / X < 0.00001
end
```

- <u>Local Procedures</u>: If a function is only used inside another, you don't need to declare the function outside.
- <u>Higher-order Programming:</u> Having first-class procedures (closure = procedure + environment). Foundation of OOP and component-based programming.
  - *Abstraction*: Creating local procedures. Any value can be wrapped inside a procedure. Not in most imperative languages (C, Pascal), because no way to encode the environment where a function is called.

        - *Genericity*: Procedures as arguments. Examples: reduce, map, filter, sum.
        - *Instantiation*: Procedures as outputs.
        - *Embedding*: Procedures in data structures. Useful in lazy evaluation, modules, and classes.
    - Lazy Evaluation: Calculate values only when needed (opposite of eager evaluation). Use cases: infinite list, streams. Haskel is lazy by default.
        - *Lazy Sieve of Eratosthenes*:

```
fun lazy {LFilter Xs F}
     case XS
     of nil then nil
     [] X|Xr then
          if {F X} then
               X|{LFilter Xr F}
          else
               {LFilter Xr F}
          end
     end
end

fun lazy {Sieve Xs}
     X|Xr = Xs in
     X | {Sieve {LFilter Xr fun {$ Y} Y mod X \= 0 end }}
end

fun {Primes} {Sieve {Ints 2}} end
```

        - *Haskell Version*:

```
ints :: (Num a) => a -> [a]
ints n = n : ints (n+1)

sieve :: (Integral a) => [a] -> [a]
sieve (x:xr) = x:sieve (filter (\y -> (y `mod` x /= 0)) xr)

primes :: (Integral a) => [a]
primes = sieve (ints 2)
```

        - *High Throughput with Lazy Buffers*: When the user asks for an item, the buffer gives the user an item and asks the producer for another.

```
fun {Buffer3 In N}
      End = thread
            {List.drop In N}
      end
      fun lazy {Loop In End}
            E2 = thread End.2 end
            In.1|{Loop In.2 E2}
      end in
      {Loop In End}
end
```

Data Types:

- Defined by a mathematical algebra (set of objects + set of operations).
- A set of operations defines Abstract Data Types (ADTs).
    - Users only interact with the abstract part of a data type (aka. operations/API).
- Type Strength:
    - *Weak*: Representational exposure. (C strings are just an array of chars that end in '\0'.)
    - *Strong Dynamic*: Variables types are known at run-time. (Oz, Python)
        - Fast prototyping. Modularity. Expressive.
    - *Strong Static*: Variables types are known at compile-time. (C++, Java)
        - Improved error-catching. Efficient. Secure. Partial program verification.
    - Not all programs are just static or dynamic. Contravariance, covariance, and type-casting.
- Type Checking: Making sure the types and operations are valid.
    - *Abstract Interpretation*: The user gives partial information (types of variables) to the compiler.
    - *Type Inference*: The compiler deduces all types implicitly. (ML, Haskell)

Secure Language:

- Secure: Well-defined and controllable properties, independent of other code.
- Capability: A language entity (ticket) that allows a user to perform an action.
    - A secure language is built on capabilities.
    - All values are capabilities (numbers, functions, strings).
    - Developed from operating system research.
- Declarative Operation: Independent, stateless, and deterministic.
    - Declarative components are good because they can be independently tested.
    - Declarative components are easier to reason about.
    - Declarative operations compose to create other declarative operations.

Functor:

- A container that holds a value.
- Map: `fmap :: (a -> b) -> f a -> f b`

- Laws:
    - `fmap id = id`
    - `fmap (f . g) = fmap f . fmap g`
- Preserve container structure.

Monads:

- A way of representing side effects.
- Binding: `(>>=) :: ma -> (a -> mb) -> mb`
- Returning: `return :: a -> ma`
- IO, Lists ([]), and Maybe are monads.
- Laws:
    - `return a >>= k             = k a`
    - `m >>= return               = m`
    - `xs >>= return . f          = fmap f xs`
    - `m >>= (\x -> k x >>= h)    = (m >>= k) >>= h`

```
lc3 = [(x,y) | x <- [1..10], y <- [1..x], x+y<= 10]
lc3' = do x <- [1..10]
          y <- [1..x]
          True <- return (x+y<=10)
          return (x,y)

lc3" = [1..10] >>=  (\x ->
          [1..x] >>= (\y ->
             return (x+y<=10) >>=
                (\b -> case b of True -> return (x,y); _ -> fail “”)))
```

Guards in list comprehensions assume that fail in the List monad returns an empty list.

- Do Syntax:
    - `do e1 ; e2        = e1 >>= \_ -> e2`
    - `do p <- e1; e2    = e1 >>= \p -> e2`

Function composition:

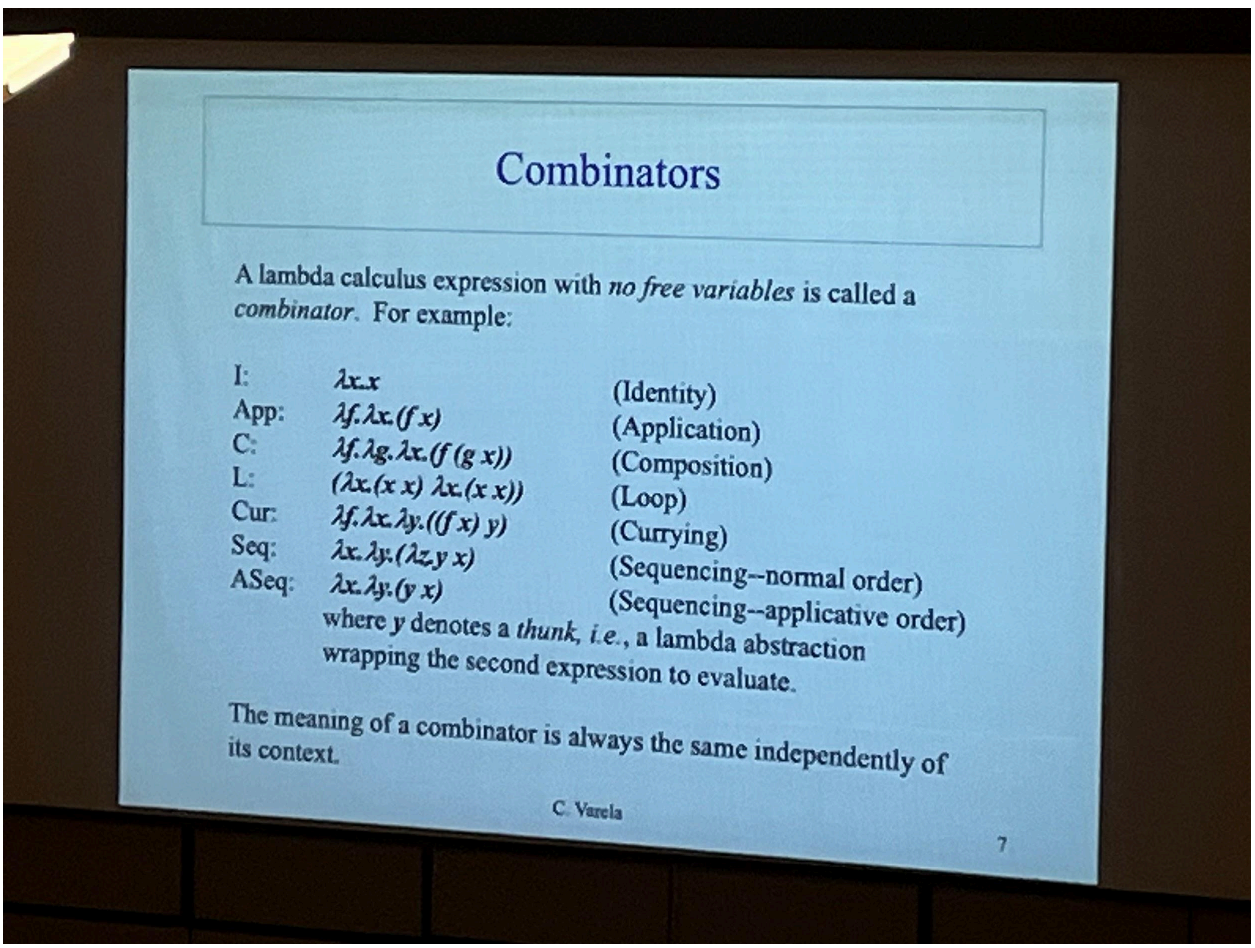
Given mathematical functions: f(x) = x^2, g(x) = x+1

Lambda Calulus:
A unified language to manipulate and reason about functions
f(x) = x^2
Labda x, x^2
Represents the same ***f*** function, except it is anonymous
To represent the function evaluation f(2) = 4, we use the following ???????

Syntax of a λ-calculus expression is as follows:
e := v → variable
| λv.e → functional abstraction
| (e*e) → function application
The semantics of a λ-calc expression is called beta-reduction
(λx, E M) ⇒ E(M/x)
Where we alpha-rename the lambda abstraction **E** if necessary to avoid capturing free vars in **M**

Note: the λ of a thing with itself is a special case and can be reduced.
*ex) (λy * y, λy * y) ⇒ λy * y*

type this up instead later



The rest of the notes are gone and can therefore not be coallated