Very not detailed summary for advanced programming

A computational system if just a system that can reason and act. A program itself is not a computational system, but a running program is. The we have computational meta-system that is a system that can analyze other systems, like debuggers and profilers.

We can talk about **reflection** that is the ability of a program to reason about itself. So a reflective system is a system that can manipulate its structure and behavior at runtime.

We have **introspection** that is the ability to reason about the structure and behavior of the program itself. And **intercession** that is the ability to change the structure and behavior of the program itself.

Important is **reification** that is a pre-condition for reflection, that is the creation of an entity that represents an entity of the object system, inside the meta-system.

Reification can be structural or behavioral. Structural is the ability of a program to reify its structure, and behavioral is the ability of a program to reify its execution.

Reification has some problems like harder compilation, slower execution and hard code.

We can talk about the referene types in java, that are used for performances. And reified types that are unique instances of the class java.lang.Class that represents the type.

An important feature that java is missing is multiple dispatch. Multiple dispatch is the ability to choose a method to call based on the runtime type of more than one of the arguments. This can be very useful in some cases.

Java is using dynamic dispatch that is the ability to choose a method to call based on the runtime type of one of the arguments. A solution for this can be using the TypeCast, that is just a cast done during the call of the method based on the type of the argument.
Of course, this is not very good when we have a lot of classes to check.

A solution is double dispatch, that is based on modifying the code and adding a method that takes the object as an argument and calls the method based on the type of the object. This is not very good because we have to modify the code a lot everytime we add a new class.

A better solution is the method call mechanism using the invoke method. This is a very good solution because we can add new classes without modifying the code. We can use the invoke method to call the method based on the type of the object, but in java is a little bit hard to handle since it requries to go up on the classes hierarchy, and we have interfaces. Moreover is based on getting some methods that should be public and that can be a problem.

About **reflection** in LISP, we can say what is LISP.

LISP is a programming language from which a lot of other programming languages took inspiration. We can have a lot of *dialetcs* that are just different versions of the same language. The most famous is Common LISP, followed by EMACS Lisp, Scheme, Clojure and Scheme.

LISP is a very powerful language that is based on the idea of having a list as the main data structure. This is very powerful because we can have a lot of things that can be represented as a list. We can have a list of lists, a list of functions, a list of variables, a list of anything.

When we define a function in LISP we are binding it to a symbol. The difference with anonymous function is that the normal function has a name, the lambda (anonymous) doesnt.

In LISP we have functions like CAR and CDR that are representing the access to the first element of the list and the rest of the list. We can have also FIRST and REST that are the same as CAR and CDR but used in Scheme.

We can have also SETCAR and SETCDR that are used to set the first element of the list and the rest of the list.

A nice thing when we talk about reflection is TRACING. Tracing is just logging the arguments and result of a function, and it's a very nice tool to inspect the actual behavior of the code.

About metaproramming, we can say that is a programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyse or transform other programs, and even modify itself while running. A common examples are *macros*, that are just a way to generate code.

In LISP we can simplify the metaprogramming using the `' , and ,@` that are used to quote, unquote and unquote-splice.

Professor started to talk about memoization that is something used a lot in dynamic programming. It's just a way to store the result of a function in a cache and use it when we need it again. It's a very useful technique to speed up the computation of some functions.

As we all know Java works converting the code into bytecode and then feeding it to the JVM. When we have a class converted in bytecode there is no way to change it. To solve this issue we have

Javassist, that is a library that allows us to modify some java classes at runtime. It's a very powerful tool that can be used to modify the behavior of a class at runtime.

It's based on creating a CtClass object that represents the class we want to modify, and then we can add fields, methods, and modify the existing ones. We can also add annotations and interfaces. It's very good because you can actually modify methods and classes while the program is running. For example, the memoization can be implemented using Javassist just by giving the class that we want to memoize and the method that we want to memoize.
Other than that, we developed even some implementation using annotation, so we could just annotate the method that we want to memoize and the library will take care of the rest. This solved the problem of the 1 single method per class that we had before.

Annotation are just some meta information that survive the compilation time and are used at runtime to get information about the program.

We even tried to implement something for taking trace of program state. We could use basic java reflection api, but we didn't have enough power to store each state of the program. So we used Javassist to modify the code and add some particular behavior, saving for each method the name of the class, the name of the modified field and the value of it. In this way, we added a lot of information to the program and we could trace the state of the program at runtime.

This was only possible using Javassist.

Talking about evaluators, we can say that an evaluator is just a program that receives an expression or a program and it computes its values.
When we speak about meta-circular evaluator, it's a program that can evaluate itself in the same language.

Talking about this,a macro is a way to generate code. It's a very powerful. The macro is actually an extension of the code that will be evaluated later.

In our evaluator made during lectures in LISP, what we did to find the values we wanted was to search inside an environment the name of the variable we were looking for. The environment is just a list of lists that contains the name of the variable and its value. We can have a lot of environments, and we can search for the variable in all of them. It will serach in a recursive way, so if it doesn't find the variable in the first environment, it will search in the second and so on.

Talking about REPL, it's a very useful tool that is used to evaluate expressions. It's just a loop that reads the input, evaluates it and prints the result. It's very useful to test the code and to see the result of the code.

Speaking of LET, it creates a new scope in which we can define some variables. We should be very careful with the scope of the variables, because if we define a variable with the same name of another variable in the upper scope, the variable in the upper scope will be shadowed.

Variables shadowing is a very common problem in programming languages, and it's very important to understand how it works. It's just the ability to define a variable with the same name of another variable in the upper scope.

About functions, we used an operator called `flet` . We would add the function to the enviroment and bind to the function itself. About the call, instead, we would search for the function inside the environment, evaluate all the arguments and extend the environment with the bindings. Then we would evaluate the body of the function with the new environment.

```
>> (flet ((+ (x y) (* x y))
    (* (x y) (+ x y)))
(+ (* 1 2) 3))
...infinite loop
```

The Lisp code enters an infinite loop due to mutual redefinitions of + and * within flet, causing circular function calls. This arises from extensions to the global environment.

def and fdef inject definitions into the environment, altering evaluation order and environment structure. Transitioning to frames from lists aids in scope separation but necessitates changes in environment operations.

The global keyword ensures access to variables defined outside function scopes, resolving issues like counters resetting within functions.

set updates variable values in the environment, even outside their scope, while evaluation order determines function argument sequence.

Implicit begin implies evaluating function arguments in a left-to-right begin block, affecting handling in let and flet.

High-order functions manipulate functions as arguments or results, exemplified by map, filter, and reduce.

The Lisp evaluator demonstrates the importance of environment structure and evaluation order in function definition and execution.

About anonymous functions, we can say that they are just functions that don't have a name. They are very useful when we have to pass a function as an argument to another function.

Dynamic scopes are built extending the environment with new bindings, then the function call creates a binding for the function arguments in this env and at the end of the call the env is deleted.

The downards funarg problem is a situation in programming in which we are passing a function to another function and the free variables of the function are affected by the environment of the function that is calling it. This can be a problem because we can have some unexpected behavior. To fix this, functions should know their environment in which the function should be evaluated.

Howhever, there is even the upward funarg problem that is the situation in which a function returns a function containing free variables. The free variables may reference another environment where the function was defined, but these variables may not be bound to the same values we are expecting, giving strange results. To fix this, we should capture the environment when the function is defined and not rely on dynamic scoping.

We got computational systems that can reason and act, with programs being a part of them when running. Meta-systems can analyze other systems too.

Reflection lets a program reason about and manipulate itself at runtime. It's split into introspection for examining structure/behavior, and intercession for modifying it.

For reflection to work, we need reification to represent program elements in the meta-system. This can make compilation harder and execution slower though.

Java uses reference types for performance, and reified types that are Class instances. But it lacks multiple dispatch to pick methods based on argument types. Dynamic dispatch only looks at one argument. Double dispatch and method invocation are workarounds, but clunky.

LISP is an influential language with dialects like Common LISP, Scheme, Clojure. Its core data structure is lists, which can flexibly represent code and data.

LISP has functions to access (CAR/CDR) and modify (SETCAR/SETCDR) list elements. Defining functions binds them to symbols.

Metaprogramming treats code as data to read, generate, analyze and modify programs, even at runtime. Macros generate code.

LISP simplifies metaprogramming with quote ('), unquote (,), and unquote-splice (,@) to control evaluation.

Memoization caches expensive function results for reuse. The Javassist library enables modifying Java bytecode and classes at runtime to implement things like memoization and tracing.

Evaluators compute expressions/programs. A meta-circular evaluator can evaluate itself.

Our LISP evaluator used environments (lists of variable bindings) to look up names. Inner environments can shadow outer ones. The flet operator creates new function scopes.

Anonymous functions are convenient as arguments. Dynamic scope extends the environment for a function call then discards it after.

The downward funarg problem is when a passed function's free variables get captured by the callee's environment. The upward funarg problem is when a returned function's free variables refer to the definer's environment. Solutions are to capture the definition environment.

Macros generate code without evaluating arguments. Hygiene avoids variable capture using auto-generated symbols (gensyms). Quasiquote (`) allows selective evaluation within macros.

Continuations represent the future of a computation, enabling advanced control flow. Nondeterminism explores multiple paths, e.g. using amb and fail operators.

The Common Lisp Object System (CLOS) supports multiple dispatch and method combination based on argument types. Generic functions have multiple methods. The class precedence list determines method specificity.