Lazy programming, and or evaluation, is the process by which evaluations and assignments are deferred until required. It is often times very useful because it allows for computation that in “greedy” or “strict” programming would either take too long, or unable to compute. It particularly is useful for evaluating recursive and potentially infinite functions/data structures. For example, having a function that returns the Nth value of the Fibonacci sequence would be extremely expensive to program using a strict language because it would try and create the Fibonacci sequence up to some arbitrary value. Assigning the same task to a lazy language would be much better because you would be able to create the sequence, and it would not be evaluated until n is defined and given to the function returning the corresponding value from the sequence. In programming, laziness is convenient, in computation, laziness is a virtue.

Some of the issues with lazy evaluation: it is often times difficult to implement efficiently. This adds a greater expense to the programmer by them having to spend more time implementing algorithms. It’s magnified since the order of evaluation can be counterintuitive which makes debugging diﬃcult. When stepping through the code, computation would be out of order, and it would be hard to find bugs. Exception handling is also essential to consider with using lazy evaluation. Since by nature of the lazy program/language expressions aren’t evaluated until the value is required, a common mistake might be to call a function inside of a Try-Catch statement, or some other exception handling tool, and the program would crash if that value wasn’t required until outside of the exception handling. For example, if the following algorithm were implemented in a lazy language:

String response = “”

try

response = getTextFromPage(url)

catch(Exception e)

Throw “error”

Print response

rather than the exception being caught by the catch statement, the program will crash when getTextFromPage(String) (which might throw a network exception, or timeOut error) is printed since getTextFromPage(url) will not have been evaluated until then. Computationally speaking, however; the more ways you can have the computer avoid doing work, the faster programs will run. This essentially is what lazy evaluations does and is the reason why lazy evaluation programs potentially run much faster than strictly programed programs. Lazy programs procrastinate computation as long as possible because often times the computer never needs to do the work at all. So for example, where a strict program might open a connection to a file, and read everything upon the program being run, a lazy program might defer that for later only after a user actually decides to have the program read that file. If you have two programs that run at the same speed, but program two only does half the amount of work, program two will always finish first.

--- Is it necessary for laziness to be built-in to the language, such as Haskell or Lazy Racket? How could you use laziness in a "non-lazy" language, such as C?

There are several languages which either have laziness built in, and many others with an option to run a program “lazily”. In order for a program to be lazy, however; it does not need to have laziness built into the language. Laziness can be implemented in the code itself. Racket, for example, can explicitly tell the complier to use lazy evaluation, and the compiler will respond by allowing evaluations to be processed as “promises” rather than defined values. The following code returns an error stating that it’s illegal to define something inside of its own definition.

#lang plai  
  
 (define natural-nums  
 (cons 0 (map add1 natural-nums))  
 )

In contrast, when the compiler is told to use lazy evaluation, the program compiles and is able to run. When asked to define “natural-nums”, the program will print out a promise.   
  
 > natural-nums  
 #<promise:natural-nums>

However, when a specific element is required of the list, the lazy program will happily comply showing that the list exists, but just has not been evaluated yet, and it then evaluates the list up to the required value.

> (third natural-nums)  
 2

--- Discuss how laziness relates to modularity.

When using lazy evaluation in a very modular program, then the program would greater control over what gets evaluated. Since function calls are only evaluated when the resulting values are needed, perhaps much of the program would never be run until the very end.

--- How does, or should, mutation relate to laziness?

--- Are there programs you have written in the past that would have been better had you understood laziness beforehand? Explain why, why not, how, etc

--- Have you used laziness before and did not realize it? Explain.

--- Is laziness "better" or more "powerful" than strictness?

This leads us to conclude that in a lot of cases, the idea of lazy evaluation is very powerful. We have to define that term though. After consulting many resources, I would define "powerful" as somewhat synonymous with "flexible". What I mean by that is, it can compute more than a strict language. Having more control over a program allows the programmer to be smarter about code. In the case of laziness, the programmer The second part of that definition gets a little grey since many would argue that any Turing-complete language can accomplish the same task as any other, but this is not necessarily a language, and more of a practice that some languages incorporate, and others need to have it coded in explicitly. And it does allow for more computation. Infinite lists are a perfect example of this. Strict languages cannot do computations on lists of an arbitrary value. They need to be given specific sizes for lists, create that list, and then perform evaluations. Laziness allows for more flexibility in that it won’t get caught in creating an infinite loop.

infinite lists, it often increases the speed of a program because it spends no time evaluating expressions that aren't needed. So computationally speaking, using lazy evaluation is much better.