Declarative Programming

What is Declarative Programming?

Programming: the process of transforming a problem into a program so that a solution of the problem can be found by a computer.

Declarative: the program is mostly a declaration of the elements of the problem and/or relationships between them. The language of the program is in contrast to *imperative languages* in which a program is an algorithm, a sequence of instructions or commands executed by a computer.

Why using Declarative Programming?

Humans express and understand relationships better than algorithms (try to execute an algorithm by hand). Declarative programs are shorter and have fewer details, hence are easier to code and understand. They inspire more confidence. Regardless of the language, often relationships must be stated before coding and tested after.

How to use Declarative Programming?

Curry: a language that joins the characterizing features of *functional* programming and *logic* programming. The language syntax and semantics will come in due time.

A problem

Before the age of computing, some of the simplest codes were sent in plain view, embedded in a long string of text. The simplest type of this embedded code is to "hide" a string of text every 'n' characters in the larger block of text. The recipient only needed to know the value of 'n', to extract the message.

You are to write a program that searches a block of text for a given string. Determine if the string is embedded somewhere, and if so, report the 'n' value.

For example,
String to search for: "Hello World"
Text to search through: "AHaealalaoa aWaoaralad"
Result: "Hello World" is found with embedding 2.

ACM Pacific NW Region Programming Contest
11 November 2000
PROBLEM E
Embedded Codes

Program Design

My interpretation is that the ciphertext can have a suffix (tail portion) that does not belong to the plaintext (otherwise the problem seems too simple). E.g. `ab` is found in `xaxbzz` with embedding 2. Other interpretations are plausible and can be discussed later.

Imperative: Try all the plausible values of n by looping over the characters of the ciphertext and plaintext. Some care is required for the control (loop boundaries) and the characters comparison (corresponding indexes).

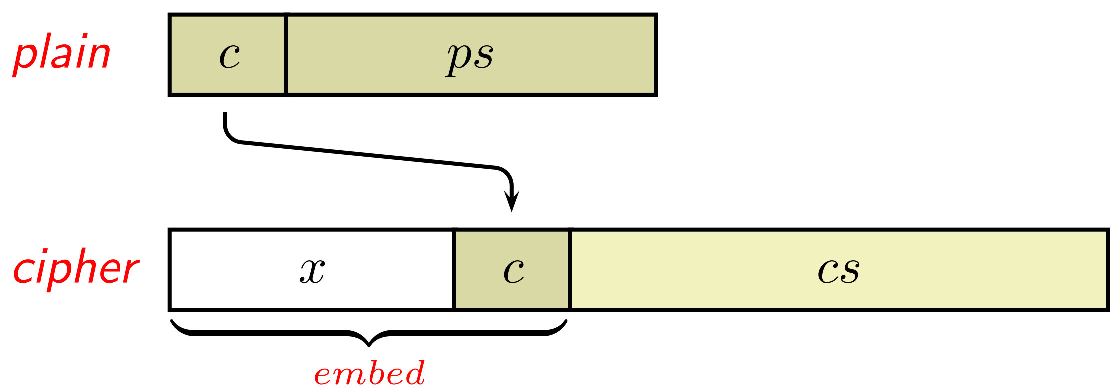
Declarative: State the relationships between embedding, ciphertext and plaintext. These relationships are already provided by the problem. Though the statement is informal, the relationships must be sufficiently precise to determine the solution(s).

Declarations

The elements of the problem

- $cipher$ is the ciphertext string
- $plain$ is the plaintext string
- $embed$ is the key of the encryption

Relationships between symbols in a graphical intutitive representation:



Relationships between symbols formalized in a system of equations:

$$\begin{cases} plain = c : ps \\ cipher = x +\!\!+ (c : cs) \\ n = length\ x + 1 \end{cases}$$

Coding

The system of equations captures the problem conditions for the first character of the plaintext. The same conditions must hold for the following characters. Thus, we name and parameterize the system of equations and recur for each character ot the plaintext.

Interestingly, the symbols "++", ":", *length* and $[\cdot]$ are library operations, hence already defined. The programmer has to code almost nothing.

In the following lectures you will learn how to encode this problem into a program. The programming language is *Curry*. The paradigm is functional logic. The computation is *narrowing*.

As a preview, this is the complete program:

```
conditions _ "" n = n
conditions (x++c:cs) (c:ps) n
   | length x + 1 == n
   = conditions cs ps n
```

As an exercise to understand and appreciate the differences between paradigms, I recommend that you solve this problem in a language you are familiar with, e.g., Java, C or python.

How does it work?

Initially, imagine that function *conditions* is called with 3 values: *cipher*, *plain*, and *embed*. If these values satisfy the conditions of the problem, the function returns *embed*. Otherwise, it reports that no value can be returned (in jargon, it fails).

For example, the call:

```
conditions "aaba" "a" 1
```

returns `1`.

Now, simplifying a little, you can replace any argument of the call with an unknown value, represented by a free variable. For example, if we want to compute the embedding:

```
conditions "aaba" "a" n where n free
```

A fundamental theorem states, simplifying a little, that if there is some value for a variable that would allow the function call to produce a result, the function execution will produce that result and also bind that value to that variable. Thus, the above call will return `1`, `2` and `4` which are also bound to variable `n`.

Observe that by choosing which arguments are values and which are free variables, function *conditions* solves different kinds of problems, e.g., decrypting or encrypting.

Homework

Code a program to solve problem E of the ACM contest. Use your favorite imperative/OO language, e.g., C, Java, Python, etc. Include both unit testing and trace of execution.

A sound way to start (as usual) is to design tests for the output of your program. This should help you understand the problem (or understand what you are not understanding) to a the degree suitable for coding a program.

Thus, code a procedure that, given some plaintext p, ciphertext c and embedding n, verifies whether p is embedded in c with embedding n. It is useful to code many tests, both positive and negative, with as many corner cases as you can.

The code for testing the program will suggest how to design the program object of this homework.

Include unit tests for each function/method of your program.