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 language of the program is mostly a declaration of the elements of the problem and/or relationships between them. The language 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. 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 encoding of 2.

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

Program Design

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

**Logic:** State the relationships between $n$, cyphertext 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

**Definition of symbols:**
- cypher is the cyphertext string
- plain is the plaintext string
- split cypher into a prefix and a suffix
- the prefix is $n$ characters long, $n > 0$
- split the prefix into some “junk” and a final character $c$
- the first character of plain is $c$

**Relationships between symbols (a system of equations):**

- $cypher = prefix ++ suffix$
- $length\ prefix = n$
- $prefix = _ ++ [c]$
- $plain = c : rest$

Graphical intuitive representation:

```
plain                  ps
---------------------
| c                   |

cipher
-----
| x     | c | cs |
   n
```
Coding

"++", ":", `length` and `[·]` are library operations.

The system 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.

```plaintext
conditions _ "" n = n
conditions cypher plain n
    | cypher == prefix ++ suffix
    & length prefix == n
    & prefix == _ ++ [c]
    & plain == c : rest
    = conditions suffix rest n
where prefix, suffix, c, rest free
```

There is a more agile syntax:

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