The cow shown laughing on the Laughing Cow® box holds, as if for earrings, two Laughing Cow® boxes each featuring a cow shown laughing and presumably — I say “presumably” because here my eyesight fails me, I don’t know about yours — holding, as if for earrings, two Laughing Cow® boxes each featuring a cow shown laughing and presumably holding... (you get the idea).

This 1921 advertising gimmick, still doing very well, is an example of a structure defined recursively, in the following sense:

### Recursive definition

A definition for a concept is recursive if it involves one or more instances of the concept itself.

“Recursion” — the use of recursive definitions — has applications throughout programming: it yields elegant ways to define syntax structures; we will also see recursively defined data structures and routines.

We may say “recursive” as an abbreviation for “recursively defined”: recursive grammar, recursive data structure, recursive routine. But this is only a convention, because we cannot say that a concept or a structure is by itself recursive: all we know is that we can describe it recursively, according to the above definition. Any particular notion — even the infinite Laughing Cow structure — may have both recursive and non-recursive definitions.

When proving properties of recursively defined concepts we will use recursive proofs, which generalize inductive proofs as performed on integers.
Recursion is \textit{direct} when the definition of $A$ cites an instance of $A$; it is \textit{indirect} if for $1 \leq i < n$ (for some $n \geq 2$) the definition of every $A_i$ cites an instance of $A_{i+1}$, and the definition of $A_n$ cites an instance of $A_1$.

In this chapter we are interested in notions for which a recursive definition is elegant and convenient. The examples include recursive routines, recursive syntax definitions and recursive data structures. We will also get a glimpse of recursive proofs.

One class of recursive data structures, the \textit{tree} in its various guises, appears in many applications and embodies the very idea of recursion. This chapter covers the important case of \textit{binary} trees.

\section*{14.1 Basic Examples}

At this point you may be wondering whether a recursive definition makes any sense at all. How can we define a concept in terms of itself? Does such a definition mean anything at all, or is it just a vicious circle?

You are right to wonder. Not all recursive definitions define anything at all. When you ask for a description of someone and all you get is \textit{“Sarah? She is just Sarah, what else can I say?”} you are not learning much. So we will have to look for criteria that guarantee that a definition is useful even if recursive.

Before we do this, however, let us convince ourselves in a more pragmatic way by looking at a few typical examples where recursion is obviously useful and seems, just as obviously, to make sense. This will give us a firm belief — little more than a belief indeed, based on hope and a prayer — that recursion is a practically useful way to define grammars, data structures and algorithms. Then it will be time to look for a proper mathematical basis on which to establish the soundness of recursive definitions.

\section*{Recursive definitions}

With the introduction of genericity, we were able to define a \textit{type} as either:

\begin{itemize}
  \item \textbf{T1} A non-generic class, such as \textit{INTEGER} or \textit{STATION}.
  \item \textbf{T2} A generic derivation, of the form $C[T]$, where $C$ is a generic class and $T$ is a type.
\end{itemize}

This is a recursive definition; it simply means, using the generic classes \textit{ARRAY} and \textit{LIST}, that valid classes are:

\begin{itemize}
  \item $\text{INTEGER}, \text{STATION}$ and such: non-generic classes, per case \textbf{T1}.
  \item Through case \textbf{T2}, direct generic derivations: \textit{ARRAY [INTEGER]}, \textit{LIST [STATION]} etc.
  \item Applying \textbf{T2} again, recursively: \textit{ARRAY [LIST [INTEGER]]}, \textit{ARRAY [ARRAY [LIST [STATION]]]} and so on: generic derivations at any level of nesting.
\end{itemize}