Designing or comprehending algorithms a human usually starts with considering a number of examples and then tries to generalize them. The goal of the researches in the inductive program synthesis is to understand and formalize this process, and to design eventually practical synthesizers.

Till the last time the inductive synthesis problems were studied basically on the recursive-theoretical level: given the sequence \( f(0), f(1), \ldots, f(m), \ldots \) of the values of a recursive function \( f \), it is necessary to restore an algorithm computing \( f \) (see surveys [1,2,3]. Many good results are obtained in this area, but unfortunately, the research on this level has not given useful ideas for the construction of practical synthesizers.

A. Biermann and R. Krishnaswamy proposed in 1976 [4] a method of inductive synthesis from examples of full computation traces. This method does not put any significant restrictions on the class of synthesizable programs, what inevitably yields an exhaustive search. Furthermore, the presentation of full computation traces is inconvenient for the user.

For practical synthesizers the class of synthesizable programs should be apparently limited - we have to look for simple inductive synthesis models applicable to reasonable problems within which the synthesis is effective and the specification of samples is convenient for a user. Therefore, the process of synthesis can be split in two stages: the selection of a model appropriate for the given problem and the synthesis itself.

Here we give a survey of models of syntactical inductive synthesis. In this approach the input information is regarded as a string of characters without any semantics, and the synthesis is based on the detection of purely syntactical analogies in a sample computation. The required program is synthesized in a form of a grammar specifying all possible computation traces (sample computations). Since, to specify a program means actually to specify all its sample computations. Such a grammar
can be considered as some nontraditional way for presenting a program scheme as a collection of all formal computation traces. Interpreting this program scheme we obtain a real program.

Of course, purely syntactical methods are restricted. For example, no syntactical algorithm is able to separate logical conditions from actions. For instance, an explanation of the sort-merge behaviour can start with

\[
\begin{align*}
    a(1) \preceq b(1) \ ? \ yes; & \quad \text{then } a(1) \rightarrow c(1); \\
    a(2) \preceq b(1) \ ? \ no; & \quad \text{then } b(1) \rightarrow c(2);
\end{align*}
\]

or with

\[
\begin{align*}
    a(1) \preceq b(1) Y; & \quad c(1): = a(1); \\
    a(2) \preceq b(1) N; & \quad c(2): = b(1);
\end{align*}
\]

here the choice of a language is dependent of a user. Any syntactical synthesis method can hardly separate logical condition in this situation. On the other hand, the advantage of syntactical methods is that they do not depend on the language chosen by a user to explain the algorithm's behaviour. This advantage is quite important, since program synthesis systems should be oriented to nonprofessional users.

The first model of inductive syntactical synthesis was developed by J.M. Barzdin [5,6]. Formalizing the notion of dots he developed a model for syntactical synthesis of programs containing only FOR-loops - the dots expressions. The basic method in this model is that of identification in sample computations fragments of arithmetical progressions.

Within this model, given an arbitrary sufficiently long sample computation, a dots expression asymptotically equivalent to the required one is synthesizable in a polynomial (from the length of the input sample) time. Thus, for the following sample computation that explains the bubble-sort algorithm in a natural way:

\[
\begin{align*}
    \text{Input } A: \text{ array } (1 \ldots 4); \\
    \text{If } A(1) \preceq A(2) \text{ then; else } A(1) \leftrightarrow A(2); \\
    \text{If } A(2) \preceq A(3) \text{ then; else } A(2) \leftrightarrow A(3); \\
    \text{If } A(3) \preceq A(4) \text{ then; else } A(3) \leftrightarrow A(4); \\
    \text{If } A(1) \preceq A(2) \text{ then; else } A(1) \leftrightarrow A(2); \\
    \text{If } A(2) \preceq A(3) \text{ then; else } A(2) \leftrightarrow A(3); \\
    \text{If } A(1) \preceq A(2) \text{ then; else } A(1) \leftrightarrow A(2); \\
    \text{Return } A,
\end{align*}
\]

we obtain the dots expression

\[
\begin{align*}
    \text{Input } A: \text{ array } (1 \ldots k); \\
    \text{\texttt{If } A(1) \preceq A(2) \text{ then; else } A(1) \leftrightarrow A(2); \quad \quad \text{\texttt{\textbullet \textbullet \textbullet}}}
    \text{If } A(k-1) \preceq A(k) \text{ then; else } A(k-1) \leftrightarrow A(k); \quad \quad \text{\texttt{\textbullet \textbullet \textbullet}}
    \text{\texttt{If } A(1) \preceq A(2) \text{ then; else } A(1) \leftrightarrow A(2); \quad \quad \text{\texttt{\textbullet \textbullet \textbullet}}}
\end{align*}
\]