A theory of rational data structures is developed, making it possible to view various models of computation as data whose processing is determined by their semantic essence and to allow for the structure induced by the computational aspect of these data. Applications of rational data structures to the theory of formal languages are described.

Modern programming and programming automation needs in specific application domains insistently present the following view: any high-level programming language is based on an algebra of data $D$ — a many-sorted algebraic system adequately reflecting the properties of the application domain. The components of the algebra $D$ are abstract data types, which may be specified in different ways: set-theoretical definitions, recursive definitions, finitely-defined algebras (generator systems with systems of defining relationships), mixed techniques, etc.

Introducing the set of names (variables, memory locations, and others) $X$, whose types are induced by the components of the algebra $D$, we obtain an information (computing) medium $I(D, X)$ as the set of all type-preserving functions from the set $X$ to the union of all the components of the algebra $D$. In this case, the set $X$ is called (main) memory, and the state of the memory $X$ is a specific element from the set $I(D, X)$.

Defining the assignment operator (simple and group assignment) as the main operator that changes the memory states and introducing the basic control compositions (multiplication, branching, naming, jump) and their derivatives (loop, switch, synchronizer, and others), we obtain the computational tools of the so-called standard programming language $SI(D, X)$ on the base algebra of data.

Extending the standard language with procedural tools and functional recursive definitions, we obtain the procedural-functional standard programming language $PFSI(D, X)$ on the base algebra of data. Of course we may use only procedural-standard tools or only functional recursive tools (or some other tools), remaining respectively within the framework of the procedural-standard or functional (or some other) programming language over the base algebra of data $D$.

Note that the syntax of sentences in the language $PFSI(D, X)$ is defined traditionally using the theory of formal grammars, and program semantics (more precisely, the semantics of the basic computational tools from which the programs are constructed) is defined using the theory of standard program schemas, discrete transducers, and fixed points.

The power of the programming language over the base algebra of data can be enhanced by adding a superstructure, specifically an algebra of data structures over $PFSI(D, X)$, and description tools for subalgebras (algebraic subsystems) of data structures. The initial base algebra of data $D$ is augmented with these subalgebras of data structures, which thus become components of a new (base) algebra of data. This process is simply a mechanism of generating new data types.

This goal cannot be achieved without first constructing a theory of data structures in which programs, as well as other models of computation, can be viewed as data having a specific structure and which expresses these specific features in terms of syntactic formalisms defining data structures.

The notion of data structure can be defined in relation to two levels of abstraction. The first level does not require formalizing the access to data structures, while the second level requires formalization of access. In other words, we assume that a second-level data structure (i.e., a data structure with access) is a couple in which the first component is a first-level data structure and the second component represents the features of access to the first component. It thus follows that a problem of primary importance is development of the concept of first-level data structure. In what follows, we will only consider first-level data structures, and the term "data structures" will be used with this qualification.
Let $A$ and $B$ be arbitrary nonempty sets. A rooted acyclic labeled graph $s = (V, E, v_0, \mu, \nu)$, where $V$ is the set of vertices, $E \subseteq V^2$ is the set of arcs, $v_0 \in V$ is the root, $\mu \subseteq V \rightarrow A$ is the vertex labeling function, $\nu \subseteq E \rightarrow B$ is the arc labeling function, is called a data structure on the sets $A$ and $B$ if for any vertex $v \in V$ the set of all children $C(v)$ is a finite set (if $(v, u) \in E$, then $u$ is called a child of the vertex $v$ and $v$ the parent of the vertex $u$). A data structure is called ordered if the set of all children of any vertex is linearly ordered.

Without attempting to justify this particular choice of definition of data structure, we will only note that acyclicity is necessary in order to satisfy the antisymmetry property of the partial order on the vertex set $V$, induced by the transitive closure of the binary relation $E$.

This leads to natural definitions of the degree of a vertex, a leaf (initial or rooted), a path in the data structure, the set of all vertices $V(u)$ reachable from the vertex $u$ in the data structure, a substructure of the data structure, etc. Also note that the definition of data structure implies that some (possibly all) arcs and vertices may be unlabeled, which creates additional possibilities.

The substructure $s$ of a data structure generated by the vertex $u$ of the data structure is the data structure $s(u) = (V(u), E(u), u, \mu(u), \nu(u))$, where $E(u), \mu(u), \nu(u)$ are the restrictions of the relation $E$ and the functions $\mu$ and $\nu$ to the corresponding sets. Note that the substructure of an ordered data structure generated by any of its vertices is again an ordered data structure.

A morphism of the data structure $s = (V, E, v_0, \mu, \nu)$ to the data structure $s' = (V', E', v_0', \mu', \nu')$ is the mapping $\varphi: V \rightarrow V'$ that satisfies the following conditions:

1) $\varphi(v_0) = v_0'$,
2) $\forall v \in V \mu(v) = \mu'(\varphi(v))$,
3) $\forall u, v \in V (u, v) \in E \Rightarrow ((\varphi(u), \varphi(v)) \in E' \Rightarrow \nu'((\varphi(u), \varphi(v))) = \nu'((\varphi(u), \varphi(v))))$.

We denote the morphism $\varphi$ of data structure $s$ to data structure $s'$ by $\varphi: s \rightarrow s'$.

The morphism $\varphi: s \rightarrow s'$ is called an isomorphism if there exits a morphism $\varphi': s' \rightarrow s$ such that $\varphi \circ \varphi' = i_s$ and $\varphi' \circ \varphi = i_{s'}$, where $i_s$ is the identity morphism $i_s: V \rightarrow V$. In this case, the data structures $s$ and $s'$ are called isomorphically equal (or isomorphic), which is denoted by $s \cong s'$.

The morphism $\varphi$ of ordered data structures $s$ and $s'$ is called an embedding of the data structure $s$ in the data structure $s'$ if $\varphi$ preserves the linear order on the corresponding sets. The notion of equal (ordered) data structures and the notion of equality of data structures are similarly defined.

Because of space limitations, we will confine the discussion to data structures up to isomorphic equality. All the results, however, remain valid for ordered data structures up to equality.

We will consider orderings on the set of data structures which can be used to develop the idea of approximation of infinite data structures by finite data structures. There are two such relations, which are dual relative to one another. The first is the isomorphic embedding relation, the second is the surmorphic embedding relation. We will now define these relations.

We say that the data structure $s$ is isomorphically embedded in the data structure $s'$ (notation, $s \trianglelefteq s'$), if there exists a substructure $s''$ of the structure $s'$ such that $s = s''$.

The morphism $\varphi: s \rightarrow s'$ is called a surjective morphism (surmorphism), and the data structure $s$ is said to be surmorphically embedded in the data structure $s'$ (notation, $s \trianglelefteq s'$), if $\varphi$ is a mapping "on" (i.e., $\forall v' \in V' \exists v \in V \varphi(v) = v'$).

THEOREM 1. The relations $\trianglelefteq$ and $\trianglelefteq$ on the set of all data structures are partial orders up to isomorphic equality.

The qualifying phrase "up to isomorphic equality" means that the conditions for the antisymmetry property imply equality of the data structures up to their isomorphism.

The isomorphic (surmorphic) limit of the sequence of data structures $\{s_n\}_{n \geq 0}$ is the exact upper bound in the relation $\leq (\trianglelefteq)$. Theorem 1 indicates that this concept is well-defined.

In what follows, we only consider isomorphic limits, using the term "a limit of a sequence of data structures" (notation, $\lim s_n$).

If a sequence of data structures has a limit, it will be called convergent. It is easy to give examples of sequences of data structures that are not convergent. It is therefore relevant to consider the question of sufficient convergence conditions for sequences of data structures.