Syntax and Semantics of the Stack Based Query Language (SBQL)¹

Kazimierz Subieta
Polish-Japanese Institute of Information Technology
subieta@pjwstk.edu.pl
Institute of Computer Science Polish Academy of Sciences
Version 2, 18 June 2010

Abstract

The Stack-Based Architecture (SBA) is a formal methodology addressing object-oriented database query and programming languages. In SBA we reconstruct query languages’ concepts from the point of view of programming languages (PLs). The approach is motivated by our belief that there is no definite border line between querying and programming; thus there should be a universal theory that uniformly covers both aspects. SBA offers a unified and universal conceptual and semantic basis for queries and programs involving queries, including programming abstractions such as procedures, functions, classes, types, methods, views, etc.

SBA assumes a semantic specification method that is referred to as *abstract implementation*. It is a kind of operational semantics where one has to determine precisely on an abstract level all the data structures that participate in query/program processing and then to specify the semantics of all the languages’ operators in terms of actions on these structures. SBA introduces three such structures that are well-known in the specification of PLs: (1) an object store, (2) an environment stack, (3) a query result stack (thus the *stack-based* architecture). These structures are fundamental for precise semantic description of everything that may happen in database query/programming languages. In particular, classical query operators, such as selection, projection, joins and quantifiers, can be generally and precisely specified using the above three abstract structures, with no references to classical database theories such relational/object algebras or calculi.

SBA introduces a model query/programming language SBQL (Stack-Based Query Language). In our intention SBQL plays the same role as relational algebra for the relational model, but SBQL is incomparably more powerful. The power of SBQL concerns a wide spectrum of data structures that it is able to serve and complete algorithmic power of querying and manipulation capabilities. At the same time, SBQL is fully precise with respect to the specification of semantics. SBQL has been carefully designed from the pragmatic (practical) point of view. We were struggling severely with parasite syntactic sugar, redundant operators and semantic reefs (when human intuitive semantics does not match machine semantics). The pragmatic quality of SBQL is achieved by orthogonality of introduced data/object constructors, orthogonality of all the language constructs, object relativism, orthogonal persistence, typing safety, introducing all the classical and some new programming

¹ This research is supported by the Polish Ministry of Science and Higher Education through the grant N N516 3755 34.
abstractions (procedures, functions, modules, types, classes, methods, views, etc.) and following commonly accepted programming languages’ principles.

SBA and SBQL are neutral to database models. SBA covers all database models that we are aware of, starting from the relational model, through XML-oriented data model, RDF-oriented data model, up to sophisticated object-oriented models with static and dynamic inheritance, collections, associations, polymorphism, etc. Our fundamental assumption is that SBA and SBQL address *data structures* rather than data models. Once we determine how particular concepts in a data model are to be mapped as abstract data structures, we can propose a corresponding subset of SBQL that will be able to handle these structures with full algorithmic universality and precision. In this way we have shifted the discussion of query language to another level: we can talk about how particular features of data structures are to be served by SBQL rather than sticking to a concrete query language with a concrete data model. For instance, when we determine how XML files will be mapped as abstract data structures, we can propose SBQL to serve these structures. In this way we achieve a unique universality, flexibility and performance optimization potential. In particular, SBQL is the first and only query language that deals with dynamic object roles and dynamic inheritance. Moreover, powerful query optimization methods that are developed for SBQL are prepared to work with such advanced features.

This report is a specification of the SBA theory and the SBQL language. It contains general observations on syntax, semantics and pragmatics of query and programming languages for object-oriented database models. General assumptions for the SBQL semantics are also presented. Then, the report deals with abstract object store models as main components of the concept of state, in particular: AS0 store model (complex objects and pointer links), AS1 store model (classes, methods and inheritance), AS2 store model (dynamic object roles and dynamic inheritance) and AS3 store model (encapsulation and information hiding). In the following the environment stack (ENVS), query results and query result stack (QRES) and function nested are introduced. These concepts form the formal basis for SBQL semantics which is defined in the operational style through abstract implementation. The core of the semantics are so-called non-algebraic operators (selection, projection, navigation, join, quantifiers, etc.), which remain algebraic operators from the relational algebra, but their general definition excludes treating them as algebraic operators. Then the syntax semantics of imperative constructs in SBQL are defined: creating objects, updating, inserting, deleting, control statements, etc. Imperative constructs use queries as expressions, there are no other expressions. On the ground of imperative constructs the syntax and semantics of procedures and methods in SBQL is defined. Next the report proposes syntax and semantics of SBQL recursive capabilities (transitive closures, fixed point equations and recursive procedures and methods). Next part of the report deals with storing and processing irregular (semi-structured) data. In Appendix 1 the report presents principles of query and programming languages and in Appendix 2. It discusses impedance mismatch, an infamous phenomenon accompanying various attempts to join query and programming languages.
Content

Preface ........................................................................................................................................ 7

1 Introduction .................................................................................................................................. 10
  1.1 General Observations on Syntax, Semantics and Pragmatics ................................................. 10
    1.1.1 Syntax .................................................................................................................................. 10
    1.1.2 Semantics ............................................................................................................................. 10
    1.1.3 Pragmatics ........................................................................................................................... 14
  1.2 Data Model and Database Schema as Components of a Query Language ......................... 14
  1.3 Abstract Syntax and Syntax-Driven Semantics ..................................................................... 17
  1.4 General Assumptions of the Semantics of Query Languages .............................................. 19
    1.4.1 Compositionality of Queries ............................................................................................. 20
    1.4.2 What is State? ...................................................................................................................... 21
    1.4.3 What is Result? ................................................................................................................... 23
    1.4.4 What is a Semantic Rule? .................................................................................................... 23
    1.4.5 Completeness of Query Languages .................................................................................... 25

2 Abstract Object Store Models .................................................................................................. 26
  2.1 AS0 Store Model: Complex Objects and Pointer Links ......................................................... 29
    2.1.1 Programming Variables and the Difference between Volatile and Persistent Data ......... 32
    2.1.2 Object Relativism ............................................................................................................... 33
    2.1.3 Collections and Structures ............................................................................................... 33
    2.1.4 Links between Objects ....................................................................................................... 34
    2.1.5 Null Values, Variants, Semi-structured Data and Types ..................................................... 35
    2.1.6 Relational Model and Nested Relational Model ................................................................. 36
    2.1.7 XML Data Model ............................................................................................................... 37
    2.1.8 Arrays in AS0 ..................................................................................................................... 39
    2.1.9 Variants of the AS0 Model ................................................................................................. 40
  2.2 AS1 Store Model: Classes and Inheritance ............................................................................ 41
  2.3 AS2 Store Model: Dynamic Object Roles and Dynamic Inheritance ................................ 46
    2.3.1 Formal Definition of the AS2 Store Model ....................................................................... 47
    2.3.2 Peculiarities of the Object Model with Dynamic Object Roles ....................................... 49
  2.4 AS3 Store Model: Encapsulation and Information Hiding .................................................... 52
    2.4.1 Orthogonal v/s Orthodox Encapsulation ......................................................................... 52
    2.4.2 Formal Basis of the AS3 Store Model ............................................................................... 54

3 Environment Stack, Query Results and Function nested .......................................................... 56
  3.1 Environment Stack in Programming Languages .................................................................... 56
  3.2 Name Binding .......................................................................................................................... 57
  3.3 Static and Dynamic Environment Stack ................................................................................ 59
  3.4 Environment Stack in the AS0 Store Model ......................................................................... 60
    3.4.1 The Concept of Binder ....................................................................................................... 62
    3.4.2 Definition of an Environment Stack ................................................................................... 62
4 SBQL Syntax and Semantics for the AS0 Model .................................................. 77
4.1 SBQL Syntax ........................................................................................................... 78
4.2 Query Evaluation Procedure eval ........................................................................... 81
4.3 Algebraic Operators ................................................................................................. 85
   4.3.1 Operators and Comparisons for Primitive Types ............................................ 86
   4.3.2 Aggregate Functions, Removing Duplicates .................................................... 87
   4.3.3 Operators and Comparisons on Collections .................................................. 88
   4.3.4 Coercions and Dereferences ........................................................................... 90
   4.3.5 Conditional Queries ....................................................................................... 91
   4.3.6 Defining Auxiliary Names ............................................................................. 91
4.4 Non-Algebraic Operators ....................................................................................... 95
   4.4.1 Procedure eval for Non-Algebraic Operators ................................................ 97
   4.4.2 Selection ......................................................................................................... 99
   4.4.3 Projection, Navigation and Path Expressions ............................................... 100
   4.4.4 Dependent/Navigational Join ...................................................................... 102
   4.4.5 Quantifiers .................................................................................................. 102
4.5 SBQL Examples and Comparisons ........................................................................ 103
   4.5.1 Comparison of Queries in SBQL and LINQ .................................................. 112
   4.5.2 Why the group by Operator is Unnecessary in Object Query Languages ...... 117
5 SBQL for AS1 and AS2 Store Models ....................................................................... 123
5.1 SBQL for AS1 Store Model .................................................................................... 123
   5.1.1 Invoking Methods in AS1 .............................................................................. 126
   5.1.2 Multiple Inheritance .................................................................................... 127
   5.1.3 Collections in the AS1 Store Model .............................................................. 128
   5.1.4 Examples of SBQL Queries for the AS1 Store Model .................................. 130
5.2 SBQL for the AS2 Store Model .............................................................................. 132
   5.2.1 Special SBQL Operators for the AS2 Model ................................................. 134
   5.2.2 Examples of SBQL Queries for the AS2 Model ............................................ 136
5.3 SBQL order by Operator and Range Queries ....................................................... 137
   5.3.1 Sorting Operator in SBQL ............................................................................ 138
   5.3.2 Empty and Multi-Valued Keys ................................................................... 140
   5.3.3 Sorting in Ascending and Descending Order ............................................... 141
   5.3.4 Alphabetic Order in Native Languages ....................................................... 141
   5.3.5 Range Queries ............................................................................................ 142
6 Imperative Constructs in SBQL ............................................................................. 144
Preface

The Stack-Based Approach (SBA) [Adam08c, Subi04, Subi09, Subi10, Subi85, Subi95, Subi95b] is a formal approach to object-oriented database query and programming languages. In SBA we reconstruct query languages’ concepts from the point of view of programming languages (PLs). The approach is motivated by our belief that there is no definite border-line between querying and programming. All attempts to establish it failed; see the relational completeness, being essentially a random, poorly motivated concept on the scale of the universality of query languages. Query languages, as facilities for database programming, absorb a lot of PLs’ functionalities: imperative programming extensions of SQL-92 (update, insert, delete, stored procedures, etc.), a new SQL standard known as SQL-99 (SQL 2008) [ANSI94, Melt93, Melt99, Melt01], Oracle PL/SQL [Oracle00], MS SQL Server Transact-SQL, the recent J2EE Hibernate tool, the Microsoft LINQ project [Linq07, Linq10], and a lot of Rapid Application Development tools. Another stream of persistent and/or polymorphic database PLs follows this line through integrating queries with programming languages. SBA is an attempt to create a unified conceptual and semantic basis for queries and programs involving queries, including programming abstractions such as procedures, functions, classes, types, methods, views, etc.

SBA and Stack-Based Query Language (SBQL) developed within SBA [Adam08c, ODRA10, Subi04, Subi09, Subi10] are neutral with respect to data models. SBA covers all the database models that we are aware of, starting from the relational model, through XML-oriented data model, RDF-oriented data model, up to sophisticated object-oriented models with static and dynamic inheritance, collections, associations, polymorphism, etc. Our fundamental assumption is that SBA and SBQL address data structures rather than specific ideological assumptions and constraints known as data models. Once one would determine how particular concepts in a data model are to be mapped as abstract data structures, we could propose a corresponding variant of SBQL that will be able to handle these structures with full algorithmic universality and precision. In particular, in the system ODRA we have implemented OCL [Warm03, OMG05, Habe07, Habe08], an OMG standard, a part of UML [OMG03, OMG07b]. This implementation rejects the obscure and doubtful “mathematical” description of OCL semantics and uses the SBA description model and the (already implemented) SBQL runtime mechanism. Hence all the implementation of OCL as an object database query language, together with optimizations and strong typing, required the effort of one person during 3 months. Similarly, on the ground of SBA the Business Process Query Language (BPQL) [Momo04, Momo05] was implemented for the commercial system Office Objects Workflow™ [Roda08]. BPQL has enjoyed a success for many commercial applications. SBA/SBQL has also been successfully applied to create a prototype of object-oriented declarative workflow management system [Dabr09, Dabr10, Dabr10b, SBQL10].

In this way the discussion of query language we have shifted on another level: we can talk how particular features of data structures are to be served by SBQL rather than sticking a concrete query language with a concrete data model. For instance, when one would determine how XML files will be mapped as abstract data structures we could propose SBQL to serve these structures. In this way we achieved the unique universality, flexibility and performance optimization potential. In particular SBQL is the first in the history query language that deals with dynamic object roles and dynamic inheritance. Moreover, powerful query optimization methods that are developed for SBQL are prepared to work with such advanced features.

---

2 A universal importer/exporter from/to XML is implemented in ODRA by Krzysztof Kaczmarski [ODRA10].
SBA can be considered as a theoretical approach with a strong and complete bridge to practice. Because development of SBA was preceded by several implementations of query languages [Icon04, Lent03, Lent06, Matt92, Schm94, Subi88, Subi90, Subi91, Subi94], it can also be considered as a practical approach resulting in a consistent and universal theory.

The design of modern and universal database PLs having querying capabilities requires methods and principles that are already acknowledged by the common practice of developing compilers and interpreters. Practically useful PLs must deal with object-orientedness, procedures and functional procedures (including recursive ones), parameter passing, various control statements, binding strategies, scoping rules, modularization, typing, etc. They should follow software engineering principles such as orthogonality, modularity, minimal definition, universality, genericity, typing safety, and clean, precise semantics.

The above issues turn out to be very severe for theoretical concepts developed in the database domain for dealing with query languages, including the relational algebra, relational calculus and formal logics. SBA is an alternative to theoretical concepts emerging on the object-orientedness wave, such as nested relational algebras [Sche86, Yazi90, Roth91], object algebras [Demu94, Scho92, Shaw89, Shaw90, Subr95, Vand91, Yu91, Poul94, Liu93, Leun93], object calculi [Grus97, Jasi98, Ried97], F-logic [Kife89], comprehensions [Bune94], structural recursion [Tann91], monoid calculus [Fega95, Grus96], functional approaches [Ship81, Laas93], etc. Careful analysis of these theoretical frames has led us to the conclusion that all of them are too limited (and sometimes totally inadequate, cf. object algebras) as the semantic basis for this kind of query languages that we intend to develop.

The SBA solution relies on adopting the classical run-time mechanism of PLs, and then, introducing to it necessary improvements. The main syntactic decision of our approach is unification of PL expressions and queries - queries remain as the only kind of PL expressions. For instance, in SBA there is no conceptual difference between expressions such as 2+2 and \((x+y)^z\) and queries such as \textit{Employee where Salary = 1000} or \((\textit{Employee where Salary} = (x+y)^z).\textit{Name} \). All such expressions/queries can be used as arguments of imperative statements, as parameters of procedures, functions or methods, as a return from a functional procedure, etc. Note that in our expressions/queries we avoid the extensive SQL-like syntactic sugar, which is non-orthogonal, sometimes illogical and makes complex queries illegible.

Concerning semantics, we focus on the classical naming-scoping-binding paradigm [Wait84]. Each name occurring in a query (or in a program involving queries) is bound to run-time programming entities (persistent data, procedures, actual parameters of procedures, local procedure objects, etc.) according to the actual scope for the name. The common PLs’ approach (that we follow in SBA) is that the scopes are organized in an environmental stack with the “search from the top” rule (thus the stack-based approach). Some extensions to the structure of stacks used in PLs are necessary to accommodate the fact that in a database we have persistent and bulk data structures and the fact that data are on a server machine, while a stack is on a client machine. Hence the stack contains data identifiers rather than data themselves (i.e., we separate the stack from a store of objects), and possibly multiple objects can be simultaneously bound to a name occurring in a query, which makes the many-data-at-a-time processing possible. The operational semantics (abstract implementation) of query operators, imperative programming constructs and procedures (functions, methods, views, etc.) is defined in terms of an abstract object store and operations on two classical stacks: environmental stack (abbreviated as ENVS) and query results stack (abbreviated as QRES).

Almost all the issues presented below are already discussed in detail in the Polish version of the SBA/SBQL book [Subi04] and in a lot of papers and reports, see SBQL web pages.
[Subi10]. Majority of language concepts and constructs presented in this report are already implemented as prototypes and commercial systems.
1 Introduction

1.1 General Observations on Syntax, Semantics and Pragmatics

Each computer language can be characterized by three aspects, known as syntax, semantics and pragmatics. Developers of computer languages, especially from the computer industry, frequently confuse these aspects, usually assuming that syntax, plus informal explanation of semantics, plus some examples completely specify a proposed language. Such an approach is represented by the ODMG and SQL standards, which specify syntax and intuitive explanations, give some examples, but essentially present no idea concerning formal semantics and (recursive) semantic interdependencies between various language constructs. This leaves a lot of room for different understanding of syntactic constructs, hence for sure will result in different, incompatible implementations. Thus, lack of formal semantics undermines the initial goals of the standards, such as portability of programs and interoperability between applications or program libraries made by independent vendors. Lack of formal semantics causes also problems with specification of a strong type checker (which must simulate the actual computations during the parse/compile time) and undermines query optimization, which have to be based on strong rules and easy reasoning concerning equivalence of different plans of actual computations for all possible database states.

1.1.1 Syntax

Syntax is determined by formal rules saying how to construct expressions of the language from the set of atomic tokens (known as alphabet). From the mathematical point of view the syntax of a language can be defined as a (usually infinite) set of all correct strings of tokens from the alphabet. Unfortunately, such definition of syntax (that can be found in popular textbooks e.g. on context-free grammars or regular expressions) is incomplete from the point of view of semantics. Semantic rules are usually associated with syntactic rules. It may happen that for the same set of correct strings there are two sets of syntactic rules A and B, but the set A is correct, while the set B is wrong. For instance, consider the SQL statement select X from Y where Z and two sets of rules A and B determining its syntax. However, the set A assumes implicitly the parentheses as in select X from (Y where Z), while the set B assumes implicitly the parentheses as in (select X from Y) where Z. Although both A and B produce the same set of SQL statements, A is correct, while B is wrong (inconsistent with the actual semantics of SQL).

Syntax is usually specified by rules of context-free grammars, perhaps with additional constraints (for instance, concerning typing). In SBA we take little attention to concrete syntax, leaving this issue for possible implementations. All our definitions will be based on abstract syntax. Perhaps, at some moment SBQL will require standardization, and at that moment the unified concrete SBQL syntax will be determined. In existing implementations of SBQL it is a bit different.

1.1.2 Semantics

Semantics determines the meaning of syntactic constructs, that is, the relationship between syntactic constructs and elements of some universe of meanings. In the popular understanding semantics addresses human minds and in this case it can be expressed in terms of a natural
language. Such semantics we can see, for instance, in UML diagrams, whose syntactic constructs are explained through more or less understandable phrases in our everyday tongue or by relationships with other (also informal) constructs. Such semantics, however, is valueless for a machine, it is not enough precise, too ambiguous and contains a lot of unspecified or poorly specified details.

The machine requires precise formal semantics. The general definition of formal semantics is not as easy as the definition of syntax because it requires the formal definition of the mentioned universe of meanings and the definition of mappings of the syntax into the universe of meanings. Such a definition is also not univocal, as it depends on who or what is the addressee of the definition. In particular, the description of semantics can be different for compiler writers and for application programmers who are/will be the users of the language. In our explanation we take the point of view of compiler or interpreter designers rather than application programmers. Obviously, this point of view requires from us to be extremely precise in specification and sensitive to all, even smallest semantic details.

In particular, we can assume that the universe of the meanings is the set of all the sequences of instructions of the Java virtual machine (JVM). The definition would be a mapping of the set of all the language expressions into the set of sequences of instructions of JVM. The definition of semantics assumes that the meaning of JVM instructions is non-definable; it is given as an axiom. Such an approach assumes that the definition of the semantics is done by the designers of a compiler or interpreter of the given language. The definition would be, however, valueless for application programmers, who rarely understand the actions of the Java virtual machine. Moreover, every team that attempts to implement Java (or other language) defines its own semantics, incompatible with the semantics of other teams.

For these reasons the semantics must be defined in abstract terms and concepts that are to be easily understood by programmers and leave no room for different interpretations by designers of compilers or interpreters of the language. The abstract terms and concepts should be much more abstract than the level of the machine code (assembler) or instructions of a Java-like virtual machine. Such semantics should address two kinds of subjects: (1) system programmers, who precisely and formally map the semantic definition into actions of the machine; (2) application programmers who will use informally the language to make applications. Formal actions of the machine and informal understanding of the semantics by application programmers must coincide. Lack of the coincidence is referred to as semantic reef. It is a property of the language that most frequently causes application programmers errors due to improper informal understanding of formal language constructs. (SQL is famous also for well known semantic reefs, in particular, with the group by clause and with null values; the issue is discussed in several papers, e.g. [Date86b, Date86c, Date92b, Subi01b, Subi96, Subi98]).

In general, precision of semantics is so important for implementation and standardization of computer languages that the general rule can be formulated as an oxymoron:

| A smallest semantic problem is the very big problem. |

Even smallest ambiguities on single bits or minor details cause that such goals as portability and interoperability become unfeasible. Unfortunately, this fact is ignored by next and next proponents of query language „standards”; thus the low quality of the standards, difficulties with consistent and entire implementation, and a lot of incompatible implementations of the same specification.

The database theory has already proposed several theoretical frameworks that are claimed to be formal bases for definition of semantics of query languages; in particular, relational
algebra (originally proposed by E.F.Codd in 1970), relational calculus and formal logic. There are also theories that are claimed to be semantic foundations for queries addressing more sophisticated database models, such as object-oriented and XML-oriented models, in particular, object algebras, an XML algebra, object calculi, variants of mathematical logic (e.g. F-logic), structural recursion, monoid calculus, etc. Our attitude to these proposals is definitely negative: in many cases we do not believe in their conceptual, mathematical and pragmatic soundness (c.f. object algebras [Subi95c]) and in all cases they offer too narrow formal basis and are inadequate to represent a lot of phenomena that occur in database models and database query/programming languages. Basically the flaws concern crippled conception, too narrow scope, inadequate mathematics and wishful thinking concerning their practical potential. Currently we believe that the Stack-Based Approach is one and the only paradigm that is acceptable as a formal basis for the description of query languages' semantics.

Our focus on formal semantics has apparently led us to mathematical methods. Unfortunately, we have no good message for mathematicians.

As a method of formal specification of computer languages, mathematics has two fundamental disadvantages:

- For real languages the full mathematical description of their semantics becomes too complex, because it must take into account all the concepts and properties of data structures (a database model) that are addressed by the language and all the language concepts, constructs and interdependencies that are welcomed by the users and designers. The complexity violates the current view on mathematical theories, which are to be based on a small number of very general concepts and aesthetic, elegant reasoning.

- Fully formal mathematical specifications would become a tight corset that practically would disallow efficient reasoning on possible changes, mutations, variants and extensions concerning the properties of data structures addressed by the language and the language constructs. These changes, mutations, variants and extensions are made due to pragmatic considerations that involve human preferences, behavior, psychology and ergonomics, hence cannot be anticipated and formalized. There are myriads of permutations of these variants and extensions and each of them could violate the given mathematical specification (thus would require new and new specifications). Moreover, a lot of these permutations are unknown: they are the subject of future inventions. Thus, mathematical specification would be the burden for the progress and evolution of the language.

Similar doubts concerning the role of mathematics in computer science are presented in [Bake92, Papa95, Tsic00]. Could we speak on formal language specification methods that are not mathematical? Mathematicians working in the computer science are trying to convince us that „formal” always means strong mathematical discipline. Fortunately, such claims are not justified and are not reasonable looking on the above fundamental disadvantages of mathematics as a formal specification method. They can be considered as attempts to make a false stereotype defending the (actually lost) position of mathematics in the computer science. The stereotype is not justified by common practice in other technical branches. For instance, the documentation of a building by an architect presents fully formal and precise specification that is sufficient to construct the building according to his/her intention. The specification uses a lot of drawings, plans, diagrams, texts and tables, but we see neither mathematical formulas nor theorems and proofs that the specification is mathematically „sound and correct”. Obviously, the specification allows for many kinds of reasoning concerning e.g. what is the optimal plan of the building construction. Similarly we can make the analogy with the car, electronic or other industries. In all these branches mathematics plays some part, but
not the major one. Still, specifications of the artifacts produced by these industries are enough formal and precise for manufacturing processes and for inferences concerning properties of products.

In no way computer languages are different in this respect. The difference concerns only the fact that computer languages are relatively young, thus methods of formal specification are not as mature as in traditional technical branches. Computers and their software are constantly changing causing the necessity of new and new formal specification methods. As in other cases of formal specification of technical artifacts we can use mathematics in all the places when mathematical concepts can be helpful for understanding. These concepts, however, will not be used in the strictly mathematical sense; in fact, we will rely only on common intuitive understanding of simple mathematical notions such as number, set, function, relation, union of sets, set containment, etc. Obviously, within this approach we are unable and we will not strive to make any mathematical proof of theorems concerning e.g. query optimization. However, we will show that our formal model can lead us to deep inferences (concerning query optimization, in particular), which are based on precise understanding of introduced concepts and relationships between them rather than on mathematical reasoning. Eventually, in every case (including mathematical proofs) implementing and practical testing of inferences is the only credible and believable proof of their correctness and usability.

Early our approach to the semantics of query languages was based on the denotational method, where each syntactic construct was associated with some abstract mathematical concept, like a function. The method is based on defining such functions by sophisticated mathematical notions known as least fixed point equations. Despite big effort to promote this approach for different software specification areas, it was totally unsuccessful, in particular, for specification of query languages’ semantics. In general, the denotational semantics is a great theory and we recommend it as beautiful exercise for everybody who is interested in top-level achievements of human intellects. Unfortunately, this way of specification was not understandable for typical designers of computer languages and deeply involves the two mentioned above fundamental disadvantages of mathematical specifications.

Currently we rely on another formal specification method known as operational semantics. The idea of the method is perhaps as old as the computer science. Many years ago it was formalized by E. Dijkstra and the A. van Wijngaarden group, but in SBA we do not refer to these old efforts. In operational semantics we have to define some machine and then, to specify the semantics of particular language constructs through operations of the machine (and through data structures that it involves). We are looking for a machine that is defined on a much more abstract level than e.g. JVM, but still is able to map formally and precisely all the language constructs. Our machine involves basic data structures that are necessary to specify the semantics of query/programming languages, that is:

- Abstract data/object store;
- Environment stack;
- Query result stack.

The method appeared to be very successful for understanding of the semantics of query languages by many people. It is sensitive to any detail of a data model that we want to consider and to any operation that we would like to introduce into a query/programming language. The operational semantics presents an abstract implementation of a language that can be directly used to make the concrete implementation in our favorite programming language. The method is also very efficient from the point of view of query optimization, i.e.
it allows for general and very deep inferences concerning how to construct an optimal query evaluation plan.

1.1.3 Pragmatics

Pragmatics of a language determines its function in interaction between humans or between a human and a machine. Pragmatics describes how to use the language in practical situations, what are the reasons for the use and what goals can be achieved. Pragmatics requires learning how to match expressions of the language to concrete real-life situations, what will be the response from the machine and how the users have to interpret the response.

A computer language should be pragmatically efficient, i.e. the language must have the potential to accomplish some important practical goals.

A computer language that is pragmatically inefficient is not a serious computer language.

In particular, one can perfectly understand syntax and semantics of a programming language, but cannot use it in pragmatically efficient way to make some usable system (the case of many so-called "theoreticians").

Pragmatics cannot be formalized. It can be explained by showing some use cases, examples, patterns, anti-patterns, best practices, wrong practices, etc. Majority of the user textbooks and documentations of languages are devoted to their pragmatics. However, the only way to teach and learn pragmatics is to use the language for concrete practical situations. By analogy, we can explain in many ways how to drive cars; however, eventually the efficient teaching requires going into a car and driving it through crowded streets.

Pragmatics of a computer language dominates over its syntax and semantics.

Pragmatics is the most important aspect of a language. Syntax and semantics are important, but only if serve the pragmatic goals of the language. The arguments in favor or against some syntactic or semantic constructs must refer to the pragmatics of the language. In particular, we reject all arguments that stem from some ideology (e.g. "the language must have sound mathematical basis"), analogy (e.g. "the language syntax must be similar to SQL") or silly associations (e.g. "a query language for XML must have the XML syntax").

In the commercial literature and documentation pragmatics is frequently confused with semantics. A typical manual presents a business intention, e.g. "Get names and salaries of employees working as programmers", and then presents a corresponding query accomplishing the intention. Such an approach to semantics is presented e.g. in the ODMG standard. Pragmatics, however, is not good as a method of specification of semantics, because it is able to present by examples some isolated semantic islands. The complete formal picture of semantics and (recursive) interdependencies between different notions and language constructs require systematic specification method, independent from examples of the use.

In the SBA and SBQL description we frequently refer to pragmatics, nevertheless our ultimate goal is the formal description of query languages' semantics.

1.2 Data Model and Database Schema as Components of a Query Language

Due to the data independence principle, the pragmatics of a query language must be extended outside the language itself. The necessary condition of pragmatic efficiency of queries is that
the programmer fully understands what the database contains and how it is organized. Because the concrete state of the database is unknown for the programmer, he/she must recognize it on some abstract level, through a database schema and business-oriented description (called ontology) that determine the business meaning of the data and services stored in the database.

To understand operations of a query language the programmer first of all must understand the database model on the level of algorithmic precision (which is required for efficient programming). Then, he/she must understand the database schema and data structures that are implied by the schema, also with algorithmic precision.

Both a data model and a concrete database schema are inevitable pragmatic parts of a query language.

In case of the relational model the situation is simple, because the model (in a pure version) introduces only named tables with named columns. Each value stored on the cross of a tuple and a column is atomic. However, simplicity of the data model and data structures that it involves is at the cost of the complexity of conceptual modeling of applications, the complexity of queries and the complexity of nesting queries within programs written in popular programming languages. Currently, the conceptual modeling tools are object-oriented, as a rule (c.f. UML). To reduce the gap between the business model and data storage model, there is a need for object-oriented database models and management systems (although the need is questioned by the commercial vendors of relational systems, for the obvious commercial reasons).

Object-oriented models introduce many useful notions that support conceptual modeling, such as complex objects with no limitations concerning size and object hierarchy levels, associations among objects, classes and inheritance among classes, behavior stored inside classes, polymorphism, and others. While for conceptual modeling these notions are (more or less) precise, attempts to map them into data structures are not trivial and not univocal. The same concepts can be mapped on many, many ways. For instance, UML class diagrams are quite understandable as a way of human thinking on some business environment. However, they are far from being precise when we try to map them to data structures. How to map inheritance and multiple inheritance? How to map associations, aggregations, qualified associations, association roles, methods, polymorphism, etc.? There is no a single answer for these questions, mapping UML class diagrams to object-oriented data structures having the property of algorithmic precision is a non-trivial research task.

The task has been solved in CORBA by IDL and mapping IDL to declarations of data structures in popular programming languages. The same is done with CORBA objects that are mapped to objects (or other structures) of the programming languages. In this way the algorithmic precision is assured. However, such an approach has disadvantages. First of all, it forces low-level programming model, in comparison to programming through query languages. The CORBA object model is also far incomplete from the point of view of modeling of business applications: it has no collections, associations, a query language and many other features (which are introduced on the level of CORBA services, with a lot of limitations and too complex notions).

The disadvantages of CORBA caused the next standardization effort known as the ODMG standard. In this framework a lot of notions that are necessary for conceptual modeling of object database applications were introduced. The standard proposes the database schema language ODL and the query language OQL. Unfortunately, the specification of object-oriented model and corresponding schema and query languages is very far from being
algorithmically precise. The standard presents also a lot of other flaws, thus we consider it as unsatisfactory for our purposes.

In an object-oriented data model and a database schema we have to reconcile several concepts and demand of different agents. Object-oriented models introduce concepts such as complex objects, object identity, classes, types, interfaces, associations, inheritance and multi-inheritance, dynamic object roles, and so on. The must present some consistent and universal whole. The model and a database schema should be clear for designers of a database, which use the schema language for determining the database structure. It should also be clear for application programmers, who have to understand the schema and objects induced by the schema with algorithmic precision. The schema must also be clear for designers of database query engine, who have to develop algorithms enabling storing, maintaining, accessing and checking data in the database, algorithms for strong type checking of queries and programs based on queries, and perhaps other algorithms. Unfortunately, understanding of object-oriented notions is very different and depends on a school, some ideology, theory or a concrete programming tool.

Many designers of database models and their schema and query languages start the job from the definition of the concept of type (c.f. the CORBA and ODMG standards, or XML technologies). Types are main components of a database schema. However, the concept of type is not obvious. Many languages have no types (e.g. Smalltalk) and nevertheless are useful. However, we advocate the view that any query/programming language should be supported by strong type checking. Types are easy when they address human minds and support conceptual modeling only. However, a type system for strong static query and program checking presents a non-trivial research issue. So far there is practically no good pattern that would be sufficiently consistent and universal. There are type theories coming from functional languages such as SML and Haskell; for our purposes these theories (although mathematical) are too restrictive. Types proposed by ODMG are inconsistent, thus non-implementable. Types introduced in the XML world (DTD, XML Schema and RDF Schema) are too limited for our purposes. After implementing a type system for SBQL we have concluded that type systems for database query/programming languages are nowadays immature and must be revisited. To have precise motivation for our type system we will introduce it at the end of our semantic considerations, when we introduce all the data structures that we want to address (on the level of algorithmic precision) and define all query operators that we want to involve into a query language.

Because in examples we need some schema language, till introducing it formally we use informal notation similar to UML. We correct it according to our needs. Below we present a simple self-explained schema built according this notation. The left part of Fig.1.1 shows the database schema in an UML-like notation. The right part shows example database structures complying with the schema, i_{10}, i_{11}, i_{12},..., i_{64} denote object identifiers. Arrows denote pointers (abstract implementation of the association between Emp and Dept objects).
Fig.1.1. Database schema and database structures complying to the schema

1.3 Abstract Syntax and Syntax-Driven Semantics

Old discussions on the syntax of programming languages were the subject of jokes, which present it as an issue for fools (c.f. the David Harel's shortest paper "do considered od" considered odder than "do considered ob"). From that time the syntax arouses some disrespect among professionals, who coined the term «syntactic sugar» to denote meaningless tokens of the language that can be arbitrarily taken by the designers. Nowadays a lot of similar jokes can be invented looking at assertions produced by the evangelists of XML. XML is a syntactic convention only. It has no semantics and no special "human-oriented" features.

Indeed, some discussions on the syntax are meaningless. However, it is not true that all such discussions are meaningless. Syntax is important for conceptual modeling (a pragmatic part of the language). Due to associations with tokens or phrases of the natural language, the meaning and the use of formal statements is easier to understand. For instance, in the SQL query:

```
select Name, Salary from Employee where Job = 'programmer'
```

the keywords `select`, `from` and `where` and characters ",", and ":=" are syntactic sugar. From the point of view of formal semantics the sugar is not essential, e.g. the same statement can be written in some hypothetical grammar as:

```
(Employee%Job:"programmer")/Name++Salary
```

Due to the sugar, the SQL statement is easier to understand and use. Too much sugar, however, combined recursively by various language constructs, may lead to clumsy queries. For instance, in SQL-92 there is possibility to insert `select` statements inside `select`, `from` and `where` clauses. In effect we can obtain constructs like

```
select ... from (select ... from ... where x = (select y from ... where ...)) where ...
```
Understanding of such nested statements can be problematic. In SQL too much sugar (plus non-orthogonality, plus lack of some operators, plus some other syntactic and semantic problems) causes that some statements are quite illegible; this is known as "SQL puzzles".

Syntax is also important as a basis for definition of semantics. Semantic definitions are independent on the syntactic sugar; hence it can be removed (or more precisely, reduced to some minimal set of abstract tokens). The syntax without sugar is referred to as abstract syntax. The syntax with all the sugar is referred to as concrete syntax. For instance, the SQL query having the concrete syntax:

```sql
select Name, DeptName from Employee, Dept where some_predicate
```

in abstract syntax can be written as:

```sql
((Employee \times Dept) \sigma some_predicate) \pi (Name ° DeptName)
```

where symbols \times, \sigma, \pi and ° denote some operators (Cartesian product, selection, projection and tuple composition, respectively). In abstract syntax we isolate all operators and syntactic constructs having independent semantic meaning (as in the above SQL example), in the right order (possibly forced by parentheses), even if some operators are not explicitly seen from the concrete syntax. In the abstract syntax it is also necessary to resolve some syntactic ambiguities that may occur in the concrete syntax. For instance, in the above SQL example commas have different meaning, depending on the context (tuple composition and Cartesian product, respectively). In our abstract syntax we map each comma to a proper abstract operator (° and \times). Operators can be written in prefix, infix and postfix style; this has no meaning for semantics. The above abstract syntax presents the infix style. In the postfix style (a.k.a. the Polish notation) it can be written as:

```sql
Employee Dept \times some_predicate \sigma Name DeptName ° \pi
```

In our presentation we prefer the infix style, which is traditional in the elementary mathematics thus more legible for the readers. The syntax will be based on context-free grammars, but for our current goals this is a secondary issue. Usually the abstract syntax is an internal part of the language definition and frequently takes the form of a special data structure known as a syntactic tree of a language expression.

In almost all computer languages semantics is syntax-driven. First, the designers of a language define its syntactic rules. Then, a semantic rule is associated with each syntactic rule. Symbols denoting operators are defined through corresponding mathematical concepts and/or some routines. Names occurring in language expressions are bound to some compile time or run time entities. Semantically, the binding means that the name fires some search in the computer or application environment for an entity (or entities) having this name. In this method syntactic rules are not arbitrary, but must reflect corresponding semantic rules. Syntactic rules are usually presented as productions of context-free grammars. Because syntactic rules are usually recursive, semantic rules must be recursive too. This method of definition we use in the description of the SBQL semantics.

The definition of semantics is much easier if the syntax is abstract rather than concrete. For query languages the most optimal case from the point of view of semantic definition is when each syntactic rule involves one operator and zero, one or at most two arguments. This allows for the most compact and most orthogonal definition of semantics (thus short implementation and much easier and more general optimization methods). SBQL is the only known query language that follows this advice. SQL and OQL present completely different syntactic style,
based on very big syntactic constructs with far context dependencies (e.g. dependency between the group by clause and select clause). The style has severe disadvantages; we do not follow it.

### 1.4 General Assumptions of the Semantics of Query Languages

We take the assumption that the semantics of any query is a (mathematical) function that maps the set of all states (basically, the database states, but not only) into the set of all query results. That is, for a given query each state is mapped into some result. More formally, let assume that Query is a set of syntactically correct queries of a given query language, State be the set of all states and Result be the set of all possible query results. Then, the semantics \(|q|\) of any query \(q \in \text{Query}\) is a function mapping State into Result:

\[\forall q \in \text{Query}: |q| : \text{State} \rightarrow \text{Result}\]

Such view on the semantics is correct for queries having purely retrieval capabilities. If a query has side-effects, i.e. it can change a state, then we must assume that each query determines a function mapping a state into a result and a new state, i.e.:

\[\forall q \in \text{Query}: |q| : \text{State} \rightarrow (\text{Result} \times \text{State})\]

Such queries may e.g. invoke a method, which makes some changes on computer, database or/and application environment.

Usually we assume the terminology in which a query always returns a result. SQL, however, has introduced queries that do not return a result, but make operations on a database. They are known as update, insert and delete clauses. According to the tradition of programming languages we will call such constructs imperative statements or instructions, avoiding the term query. If \(p\) is an imperative statement, than the semantics \(|p|\) of \(p\) can be described as a function mapping a state into a state:

\[\forall p : |p| : \text{State} \rightarrow \text{State}\]

The sets State and Result are usually different, although (as will be shown later) they are built from the same primitives. The relational model professionals worked out a stereotype known as the closure property, which can be formulated as follows: an element of State is a set of relations and an element of Result is a relation. The closure property has been considered as a necessary condition for nesting queries. This inference is based on false reasoning. After importing the closure property to an object-oriented model one can say that an element of State is a set of objects and an element of Result is a set of objects too. However, we do not buy such assertions.

For object-oriented models that we intend to deal with we conclude that the closure property is nonsense. The closure property is not a prerequisite for nesting queries, as claimed by some authors. Similarly, programming languages’ expressions act on variables but do not return variables, hence do not follow the closure property too. Obviously, expressions can be nested. In our approach we follow the simple, obvious and sound programming languages’ notions rather than speculative ideals stemming from superficial understanding of the issue.

In particular, the closure property leads to subdividing queries into object-preserving and object-generating, which is nonsense too (stemming from the previous nonsense and from inappropriate understanding of object-oriented concepts). Moreover, for relational systems, in
particular for SQL, the closure property is nonsense too. SQL is not dealing with mathematical relations, but with tables. Tables stored in the relational database have fundamentally different properties in comparison to tables returned by SQL queries. For instance, the first ones can be updated (are mutable) while the second ones cannot (are immutable). On some abstraction level we can describe them as mathematical relations (this is actually done in all the theories devoted to the relational model), but in this way a lot of semantic properties and constructs of SQL are not expressible formally, for instance, SQL updating clauses. Hence we will not follow such doubtful assertions and will define the sets \( \text{State} \) and \( \text{Result} \) according to our own perception of the issue and according to 40-years tradition of programming languages.

Now it becomes more and more clear what we have to do to define the formal semantics of a query language. We have to define formally:

- Set \( \text{Query} \) determining the abstract syntax of queries. We will not strive to define all constructs of \( \text{Query} \), showing only the basic constructs. Other constructs can be easily added by analogy with already defined ones. \( \text{Query} \) will be defined by context-free rules; each of them will be associated with a semantic rule.

- Set \( \text{State} \) determining stored data structures that are to be queried. The data structures depend on some initial ideological assumptions and constraints that the designers of a database system (or some other data repository) want to promote. In databases such ideological assumptions and constraints are referred to as data model or database model. Because there are a lot of data models and a lot of peculiarities of them, the essential question concerns a proper choice of features that are able to cover majority of them. We show that it will be possible by relatively few (currently 4) store models, and all of them have some core foundation.

- Set \( \text{Result} \) determining derived data structures that can be returned by queries in the result of their execution. To some extent, this set depends on the assumed data model too. We will define it in a quite general fashion.

- A semantic rule associated with every syntactic rule of our abstract syntax. According to the operational semantics (abstract implementation) each semantic rule will be represented by actions of some abstract machine that for the given syntactic rule performs the mapping of a state into a result.

Our approach to the formal semantics presented in the above steps will be extended to imperative constructs and to abstractions such as procedures, function and methods. We will show that the approach is an efficient theory, which is able to explain all linguistic phenomena that occur in data models and corresponding query/programming languages. This is not possible with the current theories, such as relational algebra, relational calculus, formal logic, which are able to cover only a very small subset of these phenomena and are very hard to extend to more advanced database models, such as object-oriented and XML-oriented models. The theory explains construction of query/programming languages in highly intuitive terms, allowing for efficient designing of new query/programming languages for various data models and fast direct implementation. We also intend to show that the theory is very efficient concerning reasoning on query optimization methods.

1.4.1 Compositionality of Queries

As we have noted before, because syntactic rules are recursive, semantic rules expressed by the operational machine must be recursive too. To this end we need to follow a programming
language principle known as \textit{compositionality}. The principle is the basis for recursive definitions and implementations of practically all programming languages. In short, the principle requires that the semantics of a compound statement is a function of semantics of its components. For instance, if we have a compound query \( q = q_1 \Theta q_2 \), where \( q, q_1, q_2 \) are queries and \( \Theta \) is an operator, then the semantics \( |q| \) is the result of some function \( \text{fun}_\Theta \) having \( |q_1| \) and \( |q_2| \) as arguments: \( |q| = \text{fun}_\Theta(|q_1|, |q_2|) \). Function \( \text{fun}_\Theta \) depends on the operator \( \Theta \). The compositionality property allows for orthogonal combination of operators and recursive nesting of queries. According to this principle, the designer has to determine the semantics of atomic queries, i.e. queries having no sub-queries. Then, the semantics of complex queries is build recursively from the semantics of their components. To exploit the full potential of this principle, the syntactic rules of the query language should be as short as possible, i.e. the designers should avoid big syntactic patterns and far context semantic dependencies. The principle will be applied to all constructs of our abstract syntax, including queries, imperative statements, programming abstractions and perhaps other constructs.

\subsection*{1.4.2 What is State?}

Database professionals usually assume (more or less explicitly) that in the context of query languages the concept of \textit{state} is equivalent to \textit{database state}. Moreover, it is assumed that the database state purely reflects our favorite database model (e.g. the relational model), i.e. a state is a composition of data structures that are allowed in the model (e.g. relational tables). Our view on the concept of state will be much more extended. Generally, we assume that:

\begin{quote}
A state involves all run-time entities that can influence the results returned by queries.
\end{quote}

A state involves all run-time entities whose \textit{names} can occur in queries. In particular, a database state may include not only passive database objects, but also stored procedures, views, etc., because their names can be used in queries. In object-oriented databases the concept of state must include classes and methods stored within classes. In our approach we follow the orthogonal persistence principle, which implies, in particular, that a state involves persistent and volatile data on equal rights. A state includes a \textit{metabase}, i.e. data structures that store database schema in a structural form. Entities stored in the metabase can be explicitly or implicitly referred from queries, e.g. to accomplish the method of generic programming known as \textit{reflection}. Queries may also refer to a local state of the application that the queries are nested in, e.g. to programming language variables. Hence local state of an application is also a component of the state. Queries can refer to some computer environment information, such as date, time, version of operating system, amount of free memory, etc.; hence they also augment the concept of state. Queries can also refer to some libraries (i.e. classes, procedures) that are bound to an application, thus libraries (plus states of their own variables) constitute another part of the state concept.

Query languages require also augmenting the state concept by internal data structures that store the state of query evaluation. Consider the SQL query ("get employees earning more than their boss"):

\begin{verbatim}
select * from Emp e
where e.salary > (select b.salary from Dept d, Emp b
   where e.dept# = d.dept# and d.boss = b.emp#
)
\end{verbatim}

The query contains the subquery:
\[
\text{select } b.\text{salary from Dept } d, \text{ Emp } b \text{ where } e.\text{dept#} = d.\text{dept#} \text{ and } d.\text{boss} = b.\text{emp#}
\]

having \( e \) as a free correlation variable. When the query processor evaluates this subquery, it meets \( e \) and then must find some data structure storing its value. This structure is internal to the query; it will disappear after the entire query will be evaluated. However, during evaluation of the subquery this temporal structure augments the concept of a state.

In object-oriented query languages there are more such situations when for evaluation of queries the query engine must create some data structures storing a temporary state of query evaluation. Some formal discipline concerning such temporal structures is necessary. Not surprisingly, the situation is very well recognized in programming languages and leads to the well-known concept referred to as \textit{environment stack} or \textit{call stack}. We prefer to use environment (environmental) stack, because the stack that we would like to consider will be used not only for calling functions, procedures and methods, but also for resolving some query operators (called \textit{non-algebraic}). The environment stack concept is central for our approach to query languages, thus the Stack-Based Approach (SBA).

Some professionals from the database community used to claim that the stack should not be introduced on the level of formal semantics, because it is a purely implementation notion. We strongly disagree - the definitions of some query operators cannot be formally correct without the concept. The stack is a data structure that is internally manages by the query engine. Within the description of SBA and SBQL we deal with it on the abstract rather than on the physical level. Implementation of the stack can be of course very different, especially after optimizations. The environment stack (denoted here ENVS) introduces necessary discipline for all temporary data structures that are necessary during evaluation of queries. It is a client-side structure (i.e. it is the property of a client application rather than the property of a database server) and usually it is stored and managed in the main memory.

In SBA we introduce another stack, called \textit{query result stack} (abbreviated as QRES). This stack has different role in comparison to ENVS. Although it is also a temporary data structure maintained by the query engine, we do not involve it into the concept of state. The reason is obvious: semantics of queries does not depend on the state of this stack. In the operational method that we have assumed it must be introduced explicitly, but in an equivalent denotational method this stack is unnecessary - the semantics can be explained without it. However, all methods of description of semantics of queries must refer to ENVS. Some professionals of relational databases may claim that such a stack is unnecessary for relational languages, in particular, for SQL, because they can be explained through the relational algebra or relational calculus. We don't believe them and reject all such simplistic relational ideas. The mentioned theories are far too primitive to explain all the constructs of SQL. The environment stack in some form must also occur within the SQL engine. This will be clear after we explain the semantics of basic constructs of SBQL and show examples much remaining SQL.

In general, all entities that are \textit{external} to the query engine and can influence the results returned by queries we call \textit{object store}. At some abstraction level we can make no distinction between different concepts of the store, e.g. client, server, computer environment, application environment, persistent objects, database, volatile objects, etc. All \textit{internal} entities that are to be introduced by the query engine in order to evaluate queries will be collected at the environment stack.

In conclusion, the state concept consists of two components: the state of an \textit{object store} and the state of an \textit{environment stack}. 

22
1.4.3 What is Result?

A query result concept is not as difficult as the concept of state. Because our queries cover also programming expressions, then almost any value that can be stored in the object store or can be computed from other values can be the result of a query. For instance, the query 2+2 returns 4. (The result is no surprise, but such a query can be shocking for lovers of relational theories and SQL.) However, not all values that are stored in the object store can be returned by queries. This concerns multimedia values, such e.g. as a DVD file having some gigabytes. In such cases a query should return a reference to such a value (e.g. a file name) rather than a value. In general, because according to the principle of total internal identification every object has an identifier, a query can return an identifier of an object; such a result we traditionally call reference. Reference is another fundamental concept of SBA. Queries can also introduce more complex data structures which include values, references, names, structure constructors and collection constructors. The exact definition concerning the set Result will be given later, when we will start to describe SBQL.

In the stack-based approach queries never return objects, but references to objects, perhaps within some complex data structures.

Such an assumption is the tradition in programming languages, whose expressions never return programming variables, but references to variables. Objects are data structures that are stored within the object store. Similarly to Smalltalk, we use the terminology that everything that can be stored in the store is an object, even a single value 2. Queries can refer to objects, but the result of the retrieval is volatile, it will be consumed by some agent (e.g. the print command) and after that the result no more exists. Some database theoreticians claim that there is no need to distinguish objects and references to objects, but such unification (assumed e.g. in the ODMG database standard) causes a lot of ambiguities. It worth as many as the assertion that a person named Smith and his name Smith are the same entities. The assumption that queries may return objects is the consequence of the closure property which we have totally rejected as conceptually invalid, internally inconsistent and leading to next semantically doubtful concepts.

1.4.4 What is a Semantic Rule?

As we have said before, in the operational semantics we have to define a machine that each syntactic rule associates with a semantic rule in the form of actions of the machine. We define this machine as a recursive procedure eval having a query $q$ written in an abstract syntax as an argument. The procedure eval evaluates the query $q$ and returns the result. Inside the procedure we have syntactic cases, and then, actions of the machine for particular cases. A case presents just a semantic rule subordinated to the corresponding syntactic rule. Below we present a skeleton of the procedure eval in the form of self-explained pseudo-code.

```plaintext
procedure eval( q : query )
begin
  parse ( q );  //the parser recognizes top-level subqueries in the query q

  case q is recognized as literal l :
      // a simplest query consisting of a single literal, e.g. 2, "Smith", etc.
      push the value denoted by l on the query result stack QRES;
      //the simplest semantic rule
```
The core of SBA and SBQL concerns semantic rules for queries involving operators that we call non-algebraic. We present them later. In our terminology, operator where, dot (projection, navigation), dependent (navigational) join, quantifiers, ordering and transitive closures are all non-algebraic operators. Quite surprisingly, their semantic rules are very similar and based on a simple, intuitive pattern. In the relational model some of such operators (selection, projection, join, etc.) are called algebraic. We have rejected this terminology as not fully precise from the point of view of mathematics, because it too much confuses the language and meta-language levels of mathematical description. Concerning full generality, e.g. some constructs of SQL, algebraization of these operators is possible, but the style must be fundamentally different than the style of the relational algebra. Traditionally, in mathematics operators such as quantifiers are non-algebraic (again, algebraization of them is possible, but through very advanced notions, such as cylindrical algebras, very far from primitive notions of the relational algebra). Because in SBA the semantics of quantifiers and the semantics of selection, projection and join are very similar and introduce the same notions, we unify all such operators under the name "non-algebraic". This unification is especially important for object oriented models, which are much more sophisticated and where simplistic theories coming from the relational models have failed.
1.4.5 Completeness of Query Languages

In general, the concept of "completeness" of query languages remains undefined. The relational fathers introduced so-called "relational completeness" [], but it was based on the wrong assumption that relational query languages do not require more than it is available in the relational algebra (and calculus). Even $2+2$ is not expressible in the algebra; hence this sort of "completeness" is nothing more than a random, poorly motivated point on the scale of the universality of query languages []. SQL (and SBQL) does much more than "relationally complete" languages are able to do.

Another group of researchers tried to promote so-called "Turing power", i.e. the concept of universality based on the Turing machine. Datalog [Ceri89] and OCL [Warm03] people are frequently proud that Datalog and OCL have the power equivalent to the Turing machine. Unfortunately, this is a false argument for anybody who knows mathematics. The Turing machine was invented for totally different goals than data retrieval and manipulation. It is quite easy to have the "Turing power" in a typical query and programming languages. The Basic language has the Turing power even if we remove from it 90% of its functionalities. Surely, Datalog advocates would not be happy if they hear that Datalog is as powerful as 10% of Basic. SQL has no Turing power, because it has no recursive capabilities. Still, nobody cares about that. A lot of other languages, including SBQL, have probably the Turing power, but formal proof of this fact is completely irrelevant and not interesting from the practical point of view.

Everything that we can say on universality of query languages is **pragmatic completeness**. A query language should reasonably support everything that programmers want and expect for the given application domain. The pragmatic completeness depends on the current state-of-the-art concerning expectation of programmers, thus by definition is a moving goal. If some feature of a data model or data processing is not supported, the programmers will complain that this is a disadvantage. Such complains can concern both declarations of data structures and language that are intended to process these structures.

Pragmatic completeness requires also proper qualities in the use of the language. If formulation of some queries remains puzzles (c.f. SQL) then programmers will rarely or never use them in practice. Note that programs are written once and then read 100 times. Any query that is complicated and illegible decreases the maintainability of an application. Hence proper syntax, clean semantics, orthogonality, simplicity, following typical programmer habits, etc. can be considered as factors influencing the completeness.

A query language is pragmatically incomplete if the performance of some queries is below expectation. Queries with bad performance will not be used by programmers. They should not be taken into account when considering the pragmatic completeness.

The only method to investigate pragmatic completeness is to prepare a bunch of sample queries showing the power of the language and compare this bunch to other languages. If there is an efficient and usable query in another language that is impossible to express as an efficient and usable query in our language, then our language is incomplete.

Summing up, pragmatic completeness is subjective, relative and dependent on a lot of factors that are difficult to measure. In this sense no computer language is complete, because different applications may require quite different options and nobody is able to predict all expectations of future programmers.
2 Abstract Object Store Models

As we have argued in the section devoted to syntax, semantics and pragmatics of query languages, we have to define data structures stored in an object database in an abstract way, but with the algorithmic precision. Our definition should be sufficiently universal to cover all features that can occur in object-oriented databases and XML repositories.

However current object models tend to be very complex, with many non-intuitive notions. Moreover, each object-oriented standard, programming language or database management system introduces an own object model, with very specific concepts that sound similar, but frequently have totally incompatible meaning. This concerns such popular concepts as class, inheritance, interface, type, and others. Nowadays XML technologies also contribute to this complexity. Although basically the XML model is not object-oriented, it introduces complex hierarchical objects, with a lot of own notions, especially in RDF, in XML Schema, in RDF Schema and in the family of technologies called Web Services. The ODMG standard for object-oriented databases involves many notions such as objects, literals, types, sub-types, interfaces, classes, inheritance, methods, overriding, polymorphism, collections (various kinds), structures, relationships, operations, exceptions, and others. The SQL-99 standard is even more complex, because it involves similar concepts and additionally mixes them with nested relations having a lot of peculiarities, abstract data types (ADT-s) and other features.

Unfortunately for current technologies (and fortunately for SBA) the majority of this complexity is caused by secondary features and lack of attempts to generalize, simplify the ideas, and to avoid redundant notions. For instance, one can introduce both classes and ADT-s, but conceptually the notions are the same. Similarly, the ODMG standard introduces both sets and bags as collections, but sets could be removed as a particular case of bags. If we assume the object relativism principle, then there is no need to distinguish objects and attributes. There are more such redundancies which stem from various streams of research and development, and some historical dust around the object-oriented notions that has been collected for years.

Now it is the time for cleaning the dust. For description of semantics of query languages the complexity of the underlying data model is a very negative factor. A consequence of the complexity of the object model is the complexity of a query language concerning its syntax, semantics and pragmatics. Complexity of semantics implies much more difficult implementation and optimization. In particular, due to complexity of SQL-99 many professional doubt if it is entirely implementable. Optimization of queries and programs in SQL-99 will be very challenging and in many cases impossible due to chaotic language design decisions and unknown interaction between various data structures and language’s constructs. Complexity of pragmatics leads to long documentation, extensive user manuals, long training time, long application development time and more chances for errors.

Complex semantics is also more difficult for keeping consistency. According to the conceptual closure principle, each feature of an object model must be reflected in syntax, semantics and pragmatics of a language addressing the model. The precise semantics of the language requires defining all states according to the model (the set State). The complexity of the object model causes the complexity of the set State and consequence, the complexity of definition of semantics. This leads to more difficulties during formal analysis of the semantics, decreasing the potential for query optimization, much more challenging strong type checking and much more difficult the control over completeness and mutual interaction between different constructs of the language. A complex object model causes also the
“metamodel management nightmare” (after Won Kim), that can be observed e.g. in the ODMG standard\(^3\).

Currently, the commercial world neglects or ignores the problem of the complexity of the object model and its influence on the complexity of semantics and pragmatics. The claims that for SQL-99 or ODMG OQL one can easily build a formal model are not justified at all; they belong to the marketing offices liars’ game rather than to honest and technically sound assertions. Languages are designed with no care about minimality and consistency of introduced constructs. All the commercial proposals of standards are underspecified. Holes in the specifications cause that implementations of the proposals cannot be compatible. These circumstances, together with ad hoc extensions introduced by software manufacturers, to a big extent undermine the sense of the standards.

For these reasons there is necessity to simplify object models by developing such abstractions over them that cover all the required features introduced in practical languages by minimal set of notions. We remind that just the simplicity of the relational model was the source of its success, because make it possible to reason about various properties and language constructs in intuitive and formal ways.

In contrast to the relational model, object models must be more complex for better conceptual modeling capabilities (what is just the essence of the object-orientedness). It is difficult to create a single model that would be at the same time simple and covering all the features of object models. There are also some didactic reasons: a lot of features can be explained on a very simple (but still quite universal) model, and then, next and next features can be added by generalization of this simple model. For these reasons SBA is based not on a single object store model, but on the family of models that are enumerated AS0, AS1, AS2 and AS3\(^4\); a model with a higher number introduces more sophisticated features. The list of models is of course open - there are a lot of possibilities to make variants of them or introduce new features. However, the list is complete in this sense that - according to the best of our knowledge - there is no feature or notion of currently used or proposed in practice object languages and systems that would be not covered by some of these store models. All the store models AS0, AS1, AS2 and AS3 are based on the same few formal primitives.

The basic features of the introduced store models are the following:

- **AS0 Store Model**: the simplest model that we have introduced. It covers complex hierarchical objects and pointer links between objects. It does not deal with classes, roles, inheritance, interfaces and encapsulation. AS0 covers pure relational data structures, nested relational structures, the structures implied by XML and RDF, and features that are assumed in the models referred to as semi-structured.

- **AS1 Store Model**: augments AS0 with concepts of class, static inheritance and multiple inheritance, in the most common understanding that is relevant to object databases.

- **AS2 Store Model**: augments (and a bit modifies) AS1 with the concept of dynamic object roles and dynamic inheritance (between objects).

- **AS3 Store Model**: augments AS1 or AS2 with encapsulation (subdivision of properties of objects and classes into public and private).

\(^3\) However, we disagree with Won Kim [] that this can be an argument of favor of object-relational models. In our opinion, the SQL-99 object model leads to the much more painful “metamodel management nightmare”.

\(^4\) Originally, AS0, AS1, AS2 and AS3 were denoted M0, M1, M2 and M3, correspondingly. The change was caused by a clash with the notation used by OMG for different UML (meta) models.
This family of models is rich enough to have a hope that some next conceptual feature of object-oriented artifacts will not create a new quality for semantics of defined query/programming languages. The most important is to understand the assumptions of SBA and the SBQL semantics for the simplest AS0 model. After that, it is quite easy and natural to extend the semantics to higher-order store models.

We have to warn the readers that our store models are not the same as data models. Store models are formal constructs and have almost nothing in common with concepts, ideological constraints and rhetoric, mathematical decorations, beliefs and stereotypes that are usually associated with data models. A store model is simply an abstract view on data structured stored in the database (and in other media) and is orthogonal to any ideologies such as the relational model or XML. Lack of a formally precise store model causes that the definition of semantics will be always vague and obscure; not clear for the developers and programmers of a query language engine. In our opinion, this is the case of ODMG and SQL-99 standards, which do not present the formal store model for objects explicitly, but explain them informally by other (also sometimes obscure) notions, such as types, classes, ADTs, etc.

In all the introduced store models we use the same three elementary notions:

- **Internal object identifier.** In our intention it is assigned automatically by the system and has no semantics in the external world. This is only intention, not requirement; in fact, however, we are not interested in how identifiers are generated and what internal form they have. For each object stored in the object store its internal object identifier is unique. An internal object identifier will be used to identify internally an object from the object store; in particular, it will be used as a reference or pointer to an object. As another intention we assume that the access to an object via its identifier is very fast. In programming languages internal identifier is usually a memory address or some base address plus offset. In SBA, however, we do not deal with the form or construction of identifiers and assume that the application programmer never uses it explicitly in some writable or printable form. The set of all internal identifiers we denote $I$, elements of this set will be denoted $i, i_1, i_2,...$.

- **External object name.** In contrast to internal object identifiers, external object names are explicitly assigned to objects by a database designer, a database administrator, an application programmer, or another human agent. The external object name usually bears some external business semantics of the object, is closely related to conceptual modeling of business applications and is a part of the corresponding business model e.g. determined in UML. For instance, such names can be: Person, Customer, Department, salary, deptName, job, employs, etc. In contrast to internal object identifiers, external object names need not be and usually are not unique. For instance, there may be many objects named Person and many objects named employs, on the same hierarchy level and within the same collection. In general, an object can possess many external names, assuming inheritance and the substitutability principle. For example an object named Student has also the name Person. More freedom and specific discipline in assigning names to objects is offered by the object store model AS2. The set of all external object names we denote $N$. Elements of this set will be denoted $n, n_1, n_2,...$.

- **Atomic object value.** It is the value stored within an object that from the point of the defined semantics of a query language cannot be subdivided into smaller parts. For instance, numbers 2, 3.14 and strings “Doe”, “Hello, World!” are atomic object values. We are not interested in the size of such a value - it can be a one-byte value, can contain dozens of bytes, can be a jpg file having several hundred kilobytes, or a DVD file having...
the volume of some gigabytes. Bodies of procedures, functions and methods are also atomic object values. So far we are not interested in the types of values. This issue is important, but the semantics of a query language can be explained without referring to atomic types. We will deal with the types much later, when we explain the strong type checking mechanisms. The set of all atomic object values we denote $V$, elements of this set will be denoted $v, v_1, v_2, \ldots$. In examples we use explicit self-explained literals denoting the values.

2.1 AS0 Store Model: Complex Objects and Pointer Links

Each object is represented as a triple having three components: internal object identifier, external object name and object value that can be an atomic value, an object identifier (a pointer value) or a set of objects. More formal definition:

- A triple $<i, n, v>$ is an object that we call **atomic object**. The object has the internal identifier $i$, the external name $n$ and the atomic value $v$.
- A triple $<i_1, n, i_2>$ is an object that we call **pointer object**. The object has the internal identifier $i$, the external name $n$ and internal identifier $i_2$ which is understood as a pointer (or reference) to another object.
- A triple $<i_1, n, T>$, where $T$ is a set of object, is an object that we call **complex object**. Note this rule is recursive, thus allows one to create complex objects with an unlimited number of sub-objects and many hierarchy levels.

In the model AS0 an object store is defined as a pair $<S, R>$, where $S$ is a set of objects and $R$ is set of root or start identifiers.

The set $R$ determines entry points to the object store, that is, identifiers of such objects that can be starting point for searching and navigation by a query language. Usually this concerns objects on the top hierarchy level, i.e. objects that are not nested in other objects. However, this is not a requirement. In some cases, e.g. in object store having modules or extents, just modules or extents present the top object hierarchy level, but root identifiers may include objects that are stored within the modules or extents.

The object store is also the subject of some obvious constraints:

- Each object, sub-object in an object store has a unique identifier.
- If the store contains a pointer object $<i_1, n, i_2>$ then there is an object in the store with the identifier $i_2$.
- If $i \in R$, then there is an object in the store with the identifier $i$.
- Each object in the store should be reached from objects having identifiers belonging to $R$; that is, an object should be a sub-object of the object having an identifier in $R$ (perhaps, recursively), or should be accessible by pointer objects. Objects that cannot be accessible from $R$ are not existing (they have to be removed from the store as garbage).

An example of a complex objects having three sub-objects is presented below:

$<i_5, \text{Emp}, \{<i_6, \text{Name}, \text{”Doe”}>, <i_7, \text{Sal}, 2000>, <i_8, \text{worksIn}, i_{22}>\}>$

The objects named Name and Sal are atomic objects, the objects named worksIn is a pointer object. Each object has a unique identifier.
In Fig. 2.1 we present a database schema (which is an informal notion in AS0) and one of the possible database states according to the schema. Object specifications presented in the schema are associated with cardinalities, i.e. minimal and maximal number of occurrences in an actual database. Unlimited maximal cardinality is presented by *. The cardinalities [1..1] are dropped. In Fig. 2.2 we present the same database state in a graphical notation, where objects are boxes with round corners, sub-object boxes are depicted within their parent object boxes, pointer objects are represented by arrows and root identifiers are presented within circles.

Fig. 2.1. AS0 model - a small database
The notation and representation of a database presented above is a conceptual (but algorithmically precise) view that is addressed to imagination of designers of query language interpreters (and perhaps, application programmers). It is not our intention to organize physical databases in this way. The favorite critical argument of some professionals concerning inefficiency of the organization is irrelevant for the goals that we follow. We will try to show that every known database can be conceptually perceived as a particular case of AS0-AS3.

The AS0 model does not deal with types. Many database models start considerations from defining types, but we are against such an approach. The notion of type is quite sophisticated, thus must be preceded by a lot of explanation and definitions. The issues related to types we consider very important, but we are against a simplistic approach (presented, in particular, in the ODMG standard) where types are introduced at the beginning, but chaotically and inconsistently. We plan to prepare to all the necessary things to introduce types, hence we shift explanation of types and strong type checking at the end of our discussion concerning SBA. The basic semantics of query languages can be explained without introducing types. Types, however, are necessary for majority of methods aiming query optimization. Types are also important for resolving some anomalies during processing of semi-structured data.

As we have said before, we are not interested in syntax and construction of object identifiers. However, we strongly rely on the total internal object identification principle, because a fundamental concept of the semantics of query languages defined through SBA is reference, i.e. an object identifier used as an internal object name in some place of an application program. The construction of object identifiers can be different. For instance, for object with a
fixed format an identifier of an attribute $A$ can be a pair $<OID, offset_A>$, where $OID$ is an identifier of a root object, and $offset_A$ is the number of bytes preceding the representation of $A$ in the object. Similarly, we can assume that an identifier of an attribute is a pair $<OID, attribute_name>$. However, the construction of identifiers may have the meaning for performance and pragmatic properties of a query/programming language. For instance, if one would assume that an object identifier includes the identifier of the object class, then changing the object class without changing its object identifier becomes impossible. Too long identifiers may also compromise performance.

### 2.1.1 Programming Variables and the Difference between Volatile and Persistent Data

In all store models we make no distinction between programming languages’ variables and objects. For instance, a variable $x$ defined in some programming language, having 5 as a value, in our convention will be represented as triple $<i, x, 5>$, where $i$ is an internal identifier of the variable, for instance, its address in the main memory. Similarly, we will make no distinction between the concepts of variable and objects on the principle: “unlike a variable, an object must be a member of a class”. Such a distinction presents for us no conceptual significance; it is based on some traditional terminology of particular programming languages rather than on different qualities. For instance, assuming that $x$ is of a type `integer`, we can conclude that $x$ is also a member of a class, but the class is reduced (in this time) to the type. (In general, however, we will distinguish the type and class concepts; we discuss this issue elsewhere.) We can also assume that all variables/objects have their classes, but for some of them they are empty, hence are omitted by default.

The AS0-AS3 models also do not deal with the persistence of objects. We assume the orthogonal persistence principle, i.e. the persistence property is orthogonal to all other properties of object-oriented models and is not taken into account in the specification of semantics of query/programming languages. Unlike SQL, OQL, XQuery and other database languages, for us the persistence property has practically no meaning for the definition of the semantics of query languages.

Nowadays more and more databases are stored within a main memory (and magnetic media are used as back-up devices only), thus one may ask on the difference between persistent and volatile data. Indeed, the difference concerns not a kind of media that the data is stored, but the mode of use. Data can be a property of one user (client) session and disappears when the session is terminated; therefore such a data is volatile. A persistent data is shared among many users and in typical terminology is a property of a serves rather than a client. Such data may exist even if there is no open session in the system.

Looking from this side on a query language we can conclude that a query (query execution) is a property of a client session rather than the server (with exception of some server internal processes). Hence from this point of view volatile data of the session are on the same rights as persistent server data. In conclusion, the semantics of query languages should not differentiate the access to both kinds of the data.

Of course, there should be special constructs and constraints in the languages to deal with the persistence property. For instance, there can be a programming statement which makes volatile objects persistent (i.e. causes storing them on the server) or some constraint saying that no sub-object of a volatile object can be persistent. Nevertheless, these properties are constraints we consider secondary issues concerning the definition of semantics.
2.1.2 Object Relativism

In all store models we make no distinction between objects stored on different object hierarchy levels. Subdivision of objects into simple and complex will be also secondary. We do not introduce special terminology for objects on different object hierarchy levels and objects of different complexity (such as “variable”, “object”, “composite object”, “attribute”, “sub-attribute”, “repeating attribute”, “complex attribute”, “structure”, “tuple”, “record”, “collection”, “extension”, etc.). All these notions have direct counterparts in our store models. Such distinctions may have some meaning for business-oriented external data models, but they are inessential for the definition of query languages’ semantics. Similarly to Smalltalk, an object, if it is complex, consists of objects; other terms are not necessary. Sometimes we use them to make things more clear, for instance, we sometimes use “attribute” to denote a direct sub-object of some object, but such terminology has no meaning for our formal semantics.

The object relativism has a principal meaning for simplification of the proposed languages, much simplifies a database metamodel and operations on the metamodel, increases the universality of the language and makes the syntax, semantics and pragmatics of the language more clear. These advantages in turn have a positive impact on implementation effort, development, implementation of query optimization methods and the generality of the methods. Simplification of the pragmatics results in much shorter documentations and user manuals.

Unfortunately, many proposals of object-oriented standards, languages and systems do not follow object relativism. For instance, in the ODMG standard an object attribute is “literal”, which is not an object. In many other proposals all objects must be complex, i.e. there is no possibility to consider atomic objects having no attributes. As shown above, we do not follow such crippled conceptions and constraints, because we believe they are not reasonable and lead to disadvantages concerning all the technical aspects around some object-oriented idea or product.

In this idea a module is simply an object storing other objects. One can assume some additional properties and functions of modules, e.g. they can be considered conceptual units of a database, units of reuse, units of compilation, units of exchange and substitution, units of encapsulation, but all such properties we consider orthogonal to the defined semantics of query languages.

2.1.3 Collections and Structures

In the AS0-AS3 models we assume no uniqueness of external names on any level of object hierarchy. For instance, in Fig.2.1 and Fig.2.2 name Emp are assigned to three objects and name Dept to two objects, within the Trade Dept object name location is assigned to two atomic sub-objects and within the Ads Dept object name employs is assigned to two pointer sub-objects. This is the way in which we deal with collections. Similar assumptions are taken by XML. In this way we unify several concepts related to collections, such as sets, bags, extents and repeating attributes. We also abstract from the concepts of structure, record and tuple, as known e.g. C/C++, Pascal and relational systems. For the goal of building the formal semantics of query languages such notions are secondary and can be expressed in the terms of the AS0 store model as complex objects too.

In all the store models the collection concept does not occur as a single entity having its own identity.
The apparently innocent notion of collection, as introduced in many object-oriented proposals, e.g. the ODMG standard, is inconsistent with fundamental principles of object-orientedness, such as the substitutability principle and the open-close principle. There are several signs of weakness of the collection concept. For instance, in the ODMG standard one can specify in the schema five kinds of collections, in particular, sets and bags. The same standard ensures that each object has an own identity, hence there is no two identical objects. So the question is: how can we create in the store a bag of objects? This is nonsense. Careful analysis of the standard and attempts to make all the concepts semantically clear and consistent have lead us to the conclusion that the collection concept, as a database entity having an own identity, is inconsistent and unnecessary. All that we need is the possibility to introduce many objects having the same name, just like in XML.

Obviously, in the AS0 model one can create an object named *Employees* and then, to insert into this object 10 000 objects named *Emp*. In this way we obtain the desired effect, i.e. we have created a collection having an own identity, but without introducing the collection concept explicitly. As follows from the definition of AS0, such an object can have a value being an empty set, i.e. in this way we represent an empty collection. During presentation of the model AS2 we discuss more precisely some pitfalls connected with the collection concept and the methods of avoiding them.

The collection concept, however, we will introduce in two other contexts: as a feature of the set *Result*, presenting all results returned by a query, and for a database schema language, as a type constructor. We’ll return to these notions later.

AS0 deals only with collections that are *sets*. Another interesting collection kind is *sequence*, and this kind is specific for XML. By assuming that a complex object, as a value, has a sequence of objects rather than a set of objects we obtain another store model AS0*seq* that is close to XML. It is also possible to create the model AS0*set&seq*, where complex objects can be qualified by special flags saying if the objects store sets or sequences. However, we have concluded that such a modification (although perhaps important for practice) presents no essential quality for the semantics of query languages that we have to define. Thus at this stage of our explanation we do not burden it by such secondary issues.

### 2.1.4 Links between Objects

Pointer objects assumed in AS0-AS3 are introduced to cover links among between objects. In Fig.2.1 and Fig.2.2 each pointer object *worksIn* leads to corresponding object *Dept*, and each pointer object *employs* leads to an object *Emp*. So far we take no care about the fact that such structure is redundant, because at this stage we are interested in the conceptual picture of the database state rather than in possible redundancy that we have involved. The redundancy is justified by the freedom in navigation from objects *Dept* to *Emp* and v/v.

Pointer objects can be considered as an abstract implementation of the feature that in UML is known as *association* and in the entity-relationship model (ERM), in the OMG CORBA standard and in the ODMG standard named *relationship*. (We will use *relationship*.) In ERM, UML and CORBA n-ary relationships are possible, i.e. a relationship that joins two, three or more classes (entities) and can be decorated by attributes or classes. Similarly to ODMG we have assumed in AS0-AS3 that associations can be binary only, with no decorations. We met severe difficulties in defining a store model that would introduce relationships of arbitrary arities and would be consistent and universal concerning all aspects of query/programming languages. The universality should concern not only retrieval (what is a relatively easy problem) but also all updating operations, in particular, switching an end of an association (a
role) to another object. We do not want to present here all the considerations that have led us to the conclusion. Eventually we have concluded that if an n-ary association has to be updated, it must have an identifier. Moreover, because it can be decorated by attributes, it must have methods to serve them. Hence n-ary associations should have identifiers, similarly to objects, and can be served by methods, similarly to objects too. Is it a sense to introduce two different, but very similar data structures, i.e. objects and relationships, on the level of the store model? We have concluded that there is no such sense, hence n-ary associations, n > 2, possibly with attributes, we consider equivalent to objects. In effect, we obtain objects connected by binary, non-decorated relationships, what is exactly materialized in AS0 in the form of pointer objects.

Note that the only known artifact where n-ary decorated relationships are proposed on the level of data structures (with algorithmic precision) is the CORBA Relationship Service. This proposal is extremely clumsy and it just convinced us that the idea makes little sense. Because each n-ary and decorated relationship can be easily substituted by one more class and n binary non-decorated relationships, and because unclear options for updating such n-ary relationships, we reject the idea. Our final conclusion is that n-ary relationships, where n > 2, possible decorated by attributes, present an unacceptable conceptual knot for the programmers.

### 2.1.5 Null Values, Variants, Semi-structured Data and Types

The AS0-AS3 models deal formally with the concept of null values and unions (variants). There is no requirement that objects having the same name should possess the same structure. The structure will be later constrained by types, but the type system that we have developed (and implemented) is not as restrictive as e.g. the Pascal or Java typing systems and allows for a lot of irregularities in data structures. In Fig.2.1 and Fig.2.2 the sub-object `address` is optional, hence in a particular instance of an `Emp` object can occur or not. This is the way in which we are dealing with null values.

In contrast to the relational model we do not introduce the null value concept explicitly. As argued by Date and Subieta, a special null value introduced explicitly works as a devil which is able to spoil clarity and consistency of almost all language constructs. Null values in SQL are frequently given as an example of schizophrenic inconsistency and chaotic design. A thorough discussion of the issue can be found in [Date86c, Date92b, Subi96, Subi98, Subi01b].

Our method of dealing with null values based on optional data does not lead to any inconsistency. Note that in this way we can deal with optional data on any level of object granularity. For instance, in Fig.2.1 and Fig.2.2 an optional sub-object `address` is a complex object.

In a similar way we can deal with unions or variants known from C/C++ and Pascal. Pascal introduces an additional notion of discriminator of a variant, i.e. a special value which allows during run time to recognize the actual variant and to prevent (through dynamic type checking) illegal use. In our case such a discriminator is not obligatory, but can be introduced as an ordinary sub-object. Again, one can design a special syntax of types which would make it possible to inform the type checker that a particular object has variants and that some sub-object is the discriminator of the variants. This makes it possible to shift proper type checking to run time.

As shown in Fig.2.1 (left), on the type level irregularities in data are covered by cardinality constraints. For instance, `address` is constrained by cardinalities [0..1] and `employs` is
constrained by cardinalities [1..*]. The cardinalities introduce the necessary discipline to the concept that is commonly referred to as semi-structured data. XML is frequently associated with this concept. Without such a discipline programming of large semi-structured data would be very difficult (or impossible). Eventually, the programmer must be aware what the database contains and how it is organized. This awareness must be supported on the level of algorithmic precision. The typing system involving cardinalities fulfills this goal. It will be introduced later.

The problem of specification, representation and processing of semi-structured data will be presented much later, after introducing classes, types and after defining all the necessary constructs of SBQL.

### 2.1.6 Relational Model and Nested Relational Model

The AS0 model covers the relational data structures as a particular case. In Fig.2.3 we present a relational schema, a relational table and the AS0 representation of the table.

**Relational schema:**
```
Emp( name, sal, worksIn )
```

<table>
<thead>
<tr>
<th>name</th>
<th>sal</th>
<th>worksIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doe</td>
<td>2500</td>
<td>Production</td>
</tr>
<tr>
<td>Poe</td>
<td>2000</td>
<td>Sales</td>
</tr>
<tr>
<td>Lee</td>
<td>2000</td>
<td>Sales</td>
</tr>
</tbody>
</table>

**Model AS0:**

**S - Objects:**
```
< i₁, Emp. { < i₂, name, "Doe" >,  
              < i₃, sal, 2500 >,  
              < i₄, worksIn, "Production" > } >,

< i₅, Emp. { < i₆, name, "Poe" >,  
              < i₇, sal, 2000 >,  
              < i₈, worksIn, "Sales" > } >,

< i₉, Emp. { < i₁₀, name, "Lee" >,  
              < i₁₁, sal, 2000 >,  
              < i₁₂, worksIn, "Sales" > } >
```

**R - Start identifiers**
```
I₁, I₃, I₇
```

Fig.2.3. Relational database represented in the AS0 store model

As we will show later, for such a data model SBQL queries are similar to SQL, except minor syntactic differences. Thus we can claim that SBA is also supporting the formal semantics of SQL. However, we do not strive to give the formal semantics of all the constructs of SQL. SQL is a very irregular language, with a lot of anomalies, special cases, semantics reefs and own peculiarities. Defining its formal semantics without making some order within its design makes little sense. On the other hand, SQL is a closed language, surrounded by a lot of documents, implementations and own user culture, thus any discussion on changing its features is at least 20 years too late. The SQL-99 ( SQL 2008) standards, which much extend the kinds of data structures that are addressed by the language and introduce a lot of constructs specific to programming languages, also do not present artifacts that we consider attractive to propose changes or contribute in any way. In our opinion, these standards will play the role of monuments showing (for future computer professionals) that chaotic design done by large committees, but not supported by any essential theory, leads to useless artifacts.
AS0 is richer than the relational model. A so-called natural join operator is not expressible in the pure relational model, because it is based on names of attributes, which are second-class citizens in the model, i.e. they are on the level of its informal meta-language. Such definition presents no problem assuming AS0, because external names of objects are at the same semantic level as values. Similarly, because the pure relational model is value-oriented, i.e. it does not involves identifiers of relations, tuples and values of tuples, expressing updating operations is impossible or at least requires additional notions, which would be outside the relational theory. AS0 involves no such problems because of the assumed principle of total internal identification.

The AS0 model covers also the idea on the nested relations (commonly referred to as Non-First-Normal-Form, NF$^2$) [Sche86, Yazi90, Roth91] as a particular case. AS0 makes no limitations concerning the complexity of objects and the number of object hierarchy levels. The idea how to represent in AS0 an NF$^2$ database is the same as shown in Fig.2.3. Also some models known as functional or semantic (e.g. data structures implied by the entity-relationship model) are covered by AS0.

2.1.7 XML Data Model

XML is a syntactic convention that makes it possible to unify some protocols of data exchange or to parameterize some system in some unified way. However, all the noise around XML as a new database model (or format) is in our opinion exaggeration. XML representation of data is legible for humans when the size of XML files is reasonable (say, not larger than 100 KB). For large databases storing megabytes or gigabytes of information the XML representation is nonsense. For several important reasons (security, buffering, transaction processing, indexing, query optimization, etc.) such databases should be organized according to the databases state-of-the-art, as relational, object-relational, object-oriented or a database following another paradigm.

For this reason special query languages addressing XML files have very narrow meaning. For large databases XML should not be considered as internal data representation paradigm, hence XML query languages are inapplicable. Obviously, any database application can be equipped with wrappers that accept XML files and convert the data from the files to the assumed database format. Similarly, any database application can be equipped with XML generators that convert data stored in the database to an XML file. Such wrappers and generators present no conceptual or implementation problem. Assuming this simple idea, no XML will be present inside the database, hence no XML-oriented query language is necessary.

We also note that some special requirements concerning XML query languages not always look reasonable, for instance, the requirement that an XML query should have the XML syntax. Translating this requirement to SQL, one can claim that SQL queries should be written within relational tables, what is possible, but obviously idiotic.

Taking into account the above cautions and the limited role of XML query languages, we show that the XML data model can also be considered as a particular case of AS0. We also show some features of XML that are problematic for AS0 and therefore are problematic for query languages addressing XML. In Fig.2.4 we present a simple XML file and its counterpart written as an object in the AS0 store model.
Some XML features are not covered by AS0 thus (as a rule) present problems for clean and consistent definition of XML query language.

A basic disadvantage of XML as a target of a query language is not following the total identification principle. XML objects (logical parts of an XML file) have no unique internal identifiers. Because sooner or later programs have to refer to such objects, some identification method is necessary. This is done by the Xpath language, which uses path expressions consisting of tag names. However, because XML objects may have names that are not unique at the same level of the object hierarchy, Xpath involves some tricks, for instance, determining the order number of a required object (e.g. 5-th object) or special predicates. While such methods are acceptable for relatively short XML files, they could be difficult for large parsed XML files stored in a structural database. Two potential disadvantages concern poor performance and updating anomalies (e.g. the order number n of some object A will be changed if another object A having order number m < n will be deleted).

The order of XML objects may bear information; hence in this aspect the XML data model and its AS0 representation are different. As we said before, there is no problem to create a variant of the AS0 model with complex objects storing sequences rather than sets. Alternatively, designers of XML representations may be discouraged to rely on the order of XML objects as information bearing feature. Note that ordering of objects generally decrease the query optimization potential, because some query optimization methods, e.g. indices, can lose the ordering.

XML tags may include so-called attributes, which intention is to represent meta-information rather than information. The subdivision, however, is volatile and arbitrary, thus indeed it is difficult to realize what the feature is for. In our opinion, the feature is redundant and has additional disadvantages (e.g. no DTD support), thus should be avoided. The only reasonable method is to convert attributes to some regular XML representation (and v/v). This ensures no special extensions to a query language. One of such methods is illustrated in Fig.2.5.

---

**Fig.2.4. XML file and its AS0 representation**

```
<Dept>
  <dname>Trade</dname>
  <location>Paris</location>
  <location>London</location>
  <employs>Doe</employs>
</Dept>

<Dept>
  <dname>Ads</dname>
  <location>Rome</location>
  <employs>Poe</employs>
  <employs>Lee</employs>
</Dept>
```

**S – Objects**

```
< i₁₁₇, Dept, { <i₁₈, dname, "Trade">,
  <i₁₉, location, "Paris">,
  <i₂₀, location, "London">,
  <i₂₁, employs, "Doe"} >,

< i₁₂₂, Dept, { <i₂₃, dname, "Ads">,
  <i₂₄, location, "Rome">,
  <i₂₅, employs, "Poe">,
  <i₂₆, employs, "Lee"} >
```

**R - Start identifiers**

ᵢ₁₁₇,ᵢ₁₂₂
XML has also a quite strange feature allowing for mixing atomic and complex objects. An XML object can be inserted within a string being a value of an XML object. This feature can be difficult for a query language, hence as before we suggest reduce it to the regular XML, for instance, as shown in Fig.2.6.

```
<emp>
  John Doe, born 1973
  <address> Warsaw, Sienna 5 </address>
  His salary is 2500
</emp>
```

```
<emp>
  <&info> John Doe, born 1973 </&info>
  <address> Sienna 5, Warsaw </address>
  <&info> His salary is 2500 </&info>
</emp>
```

Fig.2.6. XML file mixing strings and objects and its equivalent with no mix

### 2.1.8 Arrays in AS0

A data structure kind known as array can be modeled in AS0 by the assumption that names of objects can be numbers. For instance, array $A[1..5]$ having values subsequent values “Doe”, “Poe”, “Lee”, “Kim”, “Noe” can be represented as an AS0 complex object:

```
```

Under this assumption the access to an array element would be possible through its index being a number; for instance, $A.2$ identifies the Poe element and $A.5$ identifies the Noe element. Usually in programming languages an index of an array element can be calculated by some expression. Such a possibility was implemented in Loqis by assuming that the result of an expression that is put in square brackets is interpreted as an object name. For instance, we can write $A.[x+1]$, which for $x = 3$ returns the Kim element. In Loqis this convention was assumed as general, hence any name (including string names) can be calculated by an expression. This possibility reminds a bit a feature that is known as “reflection”. Unfortunately, it is incompatible with the strong static typing.

Such interpretation of arrays can be associated with a typing system which will assure that for each allowed index there is a corresponding array element. Alternatively, (e.g. for semi-structured data) we can allow for “thin” arrays where some indices have no corresponding elements.

Some proposals, e.g. the ODMG standard, introduce arrays and sequences as different, but a bit similar collection kinds. The idea of arrays, as presented above, is however not quite good for sequences. The difference is that sequences are always “dense”, i.e. removing an element from a sequence causes that indices of all its subsequent elements are reduced by 1; analogously for inserting an element into the sequence. As we can see from the above array example, arrays behave differently: removing an element from an array causes no changes of indices of other elements and inserting a new element into an array can be impossible (we can
only insert an element with a known index, if an element with this index is absent). Hence sequences in data structures still require a new store model different from AS0.

The above representation is good for any type of an array element. In particular, in this way we can create multi-dimensional arrays. We underline, however, that this view is abstract and concerns only the definition of query languages’ semantics. In implementation arrays can be physically represented in a more compact way.

### 2.1.9 Variants of the AS0 Model

During the work on object store models and related topics we have investigated several candidate variants that were eventually rejected as introducing limitations or the necessity of further (less intuitive) definitional elements.

**Variant 1** (from [Subi85]):

Database content is defined as the set of triples of the form \(<i, n, i_2>\) and \(<i, n, v>\). A database is defined as previously, as a pair \(<S, R>\), where \(S\) is the set of starting (root) identifiers and \(R\) is a set of the above triples. Complex objects are represented as collections of triples, e.g. a complex object named \(n\) having attributes \(a_1, \ldots, a_k\), pointer links \(p_1, \ldots, p_t\) and identifier \(i\) can be represented as:

\[
<\text{i}, \text{n}, \text{i}_1>, \ldots, <\text{i}, \text{n}, \text{i}_k>, <\text{i}, \text{n}, \text{i}_p>, \ldots, <\text{i}, \text{n}, \text{i}_p>, <\text{i}, \text{a}_1, \text{a}_1, \text{v}_1>, \ldots, <\text{i}, \text{a}_k, \text{a}_k, \text{v}_k>, <\text{i}, \text{a}_1, \text{p}_1, \text{i}_1>, \ldots, <\text{i}, \text{a}_k, \text{p}_k, \text{i}_k>
\]

The main problem with this model is that it does not determine the boundary of a complex object, thus makes problems (and the necessity of additional definitions) for updating operations, strong typing, parameter passing and other features that need encapsulated complex objects as primitive data manipulation units.

**Variant 2** (influenced by Java):

A database content is defined as a set of triples \(<i, n, \{i_1, i_2, \ldots, i_k\}>\) (representing complex objects with references to their subobjects), \(<i, n, v>\) (representing atomic object or attributes/subattributes) and \(<i, n, i_2>\) (representing reference objects). A database is defined as previously, as a pair \(<S, R>\). For instance, a complex object described previously can be represented in this formalism as:

\[
<i, n, \{i_{a_1}, \ldots, i_{a_k}, i_{p_1}, \ldots, i_{p_t}\}>, <i_{a_1}, a_1, v_{a_1}>, \ldots, <i_{a_k}, a_k, v_{a_k}>, <i_{a_1}, p_1, i_{p_1}>, \ldots, <i_{a_k}, p_k, i_{p_k}>
\]

The model makes it possible to express complex objects, but only with one object hierarchy level. To represent objects with more hierarchy levels some additional assumptions or additional elements of the formalism are required. As previously, there are doubts concerning object boundaries. In general, concerning updating of complex objects, this variant inherits disadvantages of the previous one.

Some modifications of the above variants were also investigated. In the result we have chosen AS0 as the simplest, homogeneous and not requiring too much explanations or formal elements. So far all of its variants were not introducing a new quality to formal definitions, but resulted in limitations or the necessity of defining new elements (such as flags representing boundaries of complex objects). Such add-ons are awkward considering definitions of query/programming languages’ features such as updating, strong typing and parameter passing. We have also investigated store models where (from the very beginning) objects are associated with types; thus, for instance, object names are properties of types rather than properties of objects. We abandoned this idea for two reasons. First, it is possible that our data model and a query language would be untyped (see, for instance, XQuery).
Second, types present a complex issue; thus introducing these too early in the definition, without understanding in their full complexity, might introduce limitations or inconsistencies.

2.2 AS1 Store Model: Classes and Inheritance

The concept of class is an abstraction in thinking and programming which intention is to capture both static properties of objects (i.e. their structure) and dynamic properties of objects (i.e. operations that can be performed on objects or by objects). The definition of a class addresses human minds (i.e. it supports conceptual modeling) and software engines, i.e. it allows to maintain correctly software-side data structures that are called objects. The conceptual modeling role of a class (emphasized, in particular, in UML) is very important, but has secondary meaning for the semantics of query languages. Hence we do not discuss it. We consider class as a programming entity that is associated somehow with the definition and maintenance of the data structures called objects.

In this role the concept of class has two different meanings. The first one (popular among theoreticians) has its origin in mathematics and says that from the semantic point of view a class is a set of objects (c.f. abstraction classes implied by an equivalence relation). This is wrong definition for classes understood as software units, because it does not reflects important properties of classes, such as methods. All known object-oriented programming languages, standards and systems assume (usually implicitly) another definition of the class concept, which says that:

A class is a container storing invariants (common features) for some population of objects.

Usually, object-oriented literature distinguishes two kinds of invariants that are stored within classes: typing information (i.e. names of object’s attributes together with their types) and methods (operations) that can be fired on objects (together with typing information concerning methods’ parameter and output). This is not obligatory. For instance, Smalltalk objects have no types, hence classes in Smalltalk contain no typing information. On the other hand, some object-based models do not involve methods. We can also present other invariant kinds that are stored within classes. In particular, CORBA IDL interfaces may also specify exceptions and reactions on exceptions, ODMG interfaces and classes may contain additionally relationships among objects, extents and keys. Some database models as an object invariant stored within a class include also the objects’ name.

Inheritance means that more general invariants are stored in more general classes and more specific invariants are stored in their sub-classes.

A class that is most specific for an object inherits more general invariants from its super-classes. For instance, a class FirstYearStudentClass inherits more general invariants from the class StudentClass, and yet more general invariants from the class PersonClass.

As follows from the discussion, we consider classes and types as two different semantic beings, with different roles. Typing information is necessary to strong static type checking of queries or programs acting on objects and/or for dynamic checking of objects’ structure. Classes may contain types as a particular kind of invariants, but in principle this is not obligatory (although desirable). Classes contain all the invariants that can be factored out as a common part of objects’ semantics, in particular, methods, objects’ name, etc.

Typing information is also the main component of an object interface, but again types and interfaces are different programming beings, with different semantic roles.
An interface is a programming being that has to inform the programmer how a corresponding object can be manipulated, what are constraints concerning the manipulations, and what will be the results of the manipulations.

Again, interfaces can also be defined without typing information, although the mix of typing information with interfaces we consider desirable.

Also classes and interfaces are different programming beings which should not be confused (as e.g. in the ODMG standard). Classes contain implementation, while interfaces are specifications only. Perhaps the most fundamental difference between classes and interfaces is that classes can be the subject of trade (they can be sold and bought), while interfaces cannot. This subdivision is independent on keywords that are used in a particular artifact. In the ODMG standard no classes can be specified; both keywords class and interface denote interfaces and the semantic difference between the concepts that the ODMG suggests is artificial and invented.

Interfaces are important as a pragmatic part of a query language, but essentially they have little significant for the semantics of query languages. More precisely, typing information that is a component of an interface has some influence on the semantics, but not the major one; we return to this issue later. The AS1 store model that we intend to define will not be associated with types and interfaces. Interfaces, however, is a usual way to deal with encapsulation; we return to this issue when we will consider the AS3 store model. Types and schemata (as more difficult features that usually expected) will be introduced much later.

Concerning classes, we can distinguish two forms of them:

- Classes that are parts of a source text file prepared by the programmer in some text editor. In some languages and systems (e.g. C++) this is the only form in which the classes exist. No concept of a class exists in the run-time environment; after compilation a class loses its identity, it is a part of an executable code and cannot be identified by any programming means. In such cases we will say that classes have the second-class citizenship: they exist in the source code, but they cannot be identified or manipulated during run-time.

- Classes are run-time entities that can be identified and manipulated (e.g. tested, bound, created, removed or altered) during run time. Such a class must possess its identity on the same principle as the identity for objects.

If only the first form exists, the binding of all names referring to properties of a class and occurring in a query must be done during the compilation time, i.e. a query must be compiled and linked together with the compilation and linking of a class it refers to. However, this is contrary to a basic property of queries, which in many cases must be created and interpreted during run-time. For instance, one can create and execute an ad-hoc query during operation of the database, when the whole program containing classes is already compiled and is currently executed. Because queries occur within client applications rather than within database servers, the second class citizenship means that classes are not properties of a database, which is contrary to the data independence principle.

Hence, concerning the semantics of a query language, the second class form is essential. Of course, the first form nevertheless must exist - the programmer determines classes within a source text file, including source codes of implementations of methods. After compilation such a class is converted into the second form, which is then used by the query engine.
In the AS1 store model with deal with classes in the second form only. A class is an object recorded in an object store. We associate with this object a special meaning and operations, but it will be clear after we define semantics of query operators.

ODMG essentially assumes no class stored as an object on the side of the database server. The standard presents only the first form of the class/interface representation; the same concerns the meta-model of the database (which is introduced informally and extremely obscurely). Absence of an object representing a class during run-time causes that binding of the properties of classes (e.g. names of methods) within OQL queries (which are run-time rather than compile time entities) has unknown addressee. Hence, in our opinion, the ODMG standard violates the assumptions of typical programming languages’ early binding mechanisms. Practically, this means that binding methods in OQL is non-implementable for majority of cases.

In AS1 an object store is defined as a five-tuple \(S, C, R, CC, SC\), where:

- \(S\) is a set of (perhaps nested and linked) objects, as in AS0.
- \(C\) is a set of classes. Classes are objects too.
- \(R\) is a set of identifiers of root (start) objects, as in AS0. Usually we assume that identifiers of classes are not among root identifiers.
- Relation \(CC \subseteq I_C \times I_C\) determines inheritance among classes. \(I_C \subseteq I\) denotes identifiers of classes. If \(<i_1, i_2> \in CC\), then the class identified by \(i_1\) inherits from the class identified by \(i_2\). The relation CC should not contain cycles.
- Relation \(SC \subseteq I_S \times I_C\) determines membership of objects in classes. \(I_S \subseteq I\) denotes identifiers of objects which are not classes. If \(<i_1, i_2> \in SC\), then the object identified by \(i_1\) is a member of the class identified by \(i_2\).

Each invariant stored within a class should be decorated by a flag determining its kind (a method, an object name, an export list, a trigger, etc.) but in our examples we skip these flags treating them as self-evident.

Note that the AS1 model is a superset of the AS0 model. We do not require that each object must belong to a class. It makes a sense to establish classes only in cases when they store some non-trivial invariant of a population of objects. In no such an invariant can be established, determining a class makes no sense because it does not change anything in the semantics.

In Fig.2.7 and Fig.2.8 we present a sample AS1 object store.
S – Objects

< i₁, Person, {< i₂, name, "Doe">, ... } >,
< i₅, Emp, {< i₆, name, "Poe">, < i₇, sal, 2000>, < i₈, worksIn, i₂₂>}, ... } >,
< i₉, Emp, {< i₁₀, name, "Lee">, < i₁₁, sal, 900>, < i₁₆, worksIn, i₃₃>}, ... } >

C - Classes

< i₄₀, PersonClass, {< i₄₁, age, (...the code of the method age)>, ... other invariants of the PersonClass... } >,
< i₅₀, EmpClass, {< i₅₁, changeSal, (... the code of the method changeSal...), < i₅₂, netSal, (... the code of the method netSal ...), ... other invariants of the EmpClass... } >

R - Start identifiers

i₁, i₅, i₉

CC - Inheritance among classes

< i₅₀, i₄₀>

SC - Membership of objects within classes

< i₁, i₄₀>, < i₅, i₅₀>, < i₉, i₅₀>

Fig.2.7. Example of an AS1 object store
As before, in the graphical representation the identifiers of root objects are within circles. Identifiers of classes are not among root objects; hence we assume that queries cannot directly refer to classes. For some purposes, e.g. administration of the store, we can imagine that class objects such as `PersonClass` and `EmpClass` can be manipulated, e.g. removed or altered. Under this assumption, identifiers of them should belong to root identifiers, but perhaps only in the special administrative mode. An arrow with a big white triangle end denotes inheritance (CC) and thick gray arrows denote membership of objects within classes (SC). Classes contain methods, together with their compiled implementation. Methods are understood as procedures with some specific scoping rules; this will be explained later. To simplify the picture in this representation we do not present formal parameters of the methods; this feature will also be introduced later. Note that this is an abstract view; the relations CC and SC can be implemented physically in many ways. For instance, the SC relation can be implemented by containers storing objects belonging to particular classes. So far we also say nothing about how the relations CC and SC will be used by the query execution engine; this will be considered later. Our intention is to define the abstract store in which such relations can be recorded.

The AS1 store model covers also multiple inheritance and the possibility that one object is a direct member of more than one class. We allow for pairs \( <i_1, i_2> \), \( <i, i_1> \) \( \in \) CC such that \( i_1 \neq i_2 \); similarly for SC. Such situations can be handled by the defined query engine with no difficulties. There are cases when multiple inheritance and multiple membership are reasonable, thus we do not forbid them.
The AS1 model is an abstraction over the most popular models of object-oriented programming languages, modeling tools and database systems. It allows for accomplishing the substitutability principle. The principle is quite easy to implement through a proper name binding algorithm within the query execution engine. Although the model seems to be simple and natural, it leads to problems concerning, in particular, multiple inheritance and repeating inheritance. In particular, if we assume that classes A and B are developed independently, they may contain methods having the same name and type and we define a class C that inherits from A and B, then one of the two fundamental principles of object-orientedness - the substitutability principle or the open-close principle - must be violated. The AS1 store model has also severe disadvantages as a database model, because (as we have argued before) substitutability is in contradiction with the concept of collections of objects and the open-close principle. For these reasons we introduce the AS2 store model, which is the cure for all the conceptual shortcomings of AS1.

2.3 AS2 Store Model: Dynamic Object Roles and Dynamic Inheritance

The idea of dynamic object roles\(^5\) is simple and natural. It assumes that every real or abstract entity during its life can acquire and lose many roles without changing its identity. The roles appear during the life of a given object, they can exist simultaneously, and they can disappear at every moment. For example, a certain person can at the same time be a student, a worker, a patient, a club member, etc., Fig.2.9. Similarly, a building can be an office, a house, a magazine, etc.

![Fig.2.9. Roles of a person object](image)

Typical object models have a possibility to express static properties, e.g., the fact that a student is a person. However, it is more precise to say that a person becomes a student for some time and later he or she terminates the student role. Moreover, some person at the same time can be a student two or more times. Similarly, a person may become an employee, a patient, etc. only for some time.

The concept of dynamic object roles assumes that an object is associated with other objects (subobjects), which are modeling its roles. Object-roles cannot exist without their parent object (in Fig.2.9, without the Person object). Deleting an object causes deleting all of its roles. Roles can exist simultaneously and independently. A role can have its own additional

\(^5\) The term role is overloaded. Dynamic object roles have nothing in common with association roles, as introduced in UML. Our understanding of the term is also different from the role of a person in some software development methodologies and in the workflow terminology. Our role is also different from the concept of role, as understood in the Object Role Modeling (ORM) developed by Terry Halpin and other authors.
attributes and methods. It is normal that two roles can contain attributes and methods with the same names, and this does not lead to any conflict. This is a fundamental difference in comparison to the concept of multiple inheritance.

Relationships (associations) between objects can connect not only objects with objects, but also objects with roles and roles with roles. For example, a relationship worksIn connects an Emp role with a Dept object. This makes the referential semantics clean in comparison to the traditional object models. Roles can be further specialized as sub-roles, sub-sub-roles, etc. For example, the specialization of a role Club_Member can be a role Club_President.

The role concept requires introducing composite objects with special structure and semantics. The structure should be supported by proper generic operations. In this section we describe the structure formally, as the AS2 store model. In next chapters we precisely describe how such a feature can be involved into a query/programming language.

Dynamic object roles have had for several years the reputation of a notion on the brink of acceptance. There are many papers advocating the concept, however, many researchers consider applications of the concept not sufficiently wide to justify the extra complexity of conceptual modeling facilities. The concept is totally neglected on the implementation side - as far as we know none of popular object-oriented programming languages or database systems introduces it explicitly. It is implemented in prototypes, e.g. the Fibonacci and Loqis systems. Some authors assume a tradeoff, where the role concept is the subject of special design patterns applied both on the conceptual modeling and the implementation sides. We doubt if all aspects of the concept can be reflected in this way and if indeed such design patterns simplify the programmers’ job.

The low popularity of the notion is caused by the already established object-oriented principles, especially in programming languages. The basic assumption is that objects conform to the substitutability principle, which seems to be very natural, but on the other hand leads to anomalies, which are evident in the case of multiple, multiple-aspect and repeating inheritance. Another assumption, which impedes the popularity of dynamic roles, is strong static (polymorphic) typing, which in case of dynamic roles must be redesigned.

We will try to convince the reader that the mentioned impediments of wide usage of roles can be avoided. We show that the dynamic object roles are useful both for conceptual modeling and implementation. The concept could much facilitate modeling tools such as UML and could be an important paradigm on object databases built e.g. in the spirit of the ODMG standard. Moreover, dynamic object roles allow us to avoid some limitations, pitfalls and inconsistencies in object-oriented database models.

2.3.1 Formal Definition of the AS2 Store Model

In our definition of the AS2 store model we accept all the definitions of the AS0 and AS1 model and extend them by few new concepts. In AS2 an object store is defined as a six-tuple \( <S, C, R, CC, SC, SS> \), where:

- \( S \) is a set of (perhaps nested and linked) objects, as in AS0.
- \( C \) is a set of classes, as in AS1.
- \( R \) is a set of identifiers of root (start) objects, as in AS0.
- Relation \( CC \subseteq \mathbb{I}_C \times \mathbb{I}_C \) determines static inheritance among classes, as in AS1.
- Relation \( SC \subseteq \mathbb{I}_S \times \mathbb{I}_C \) determines membership of objects in classes, as in AS1.
• New relation $SS \ I S \times I S$ determines \textit{dynamic inheritance} among objects. If $<i_1, i_2>$ $\in SS$, then the object identified by $i_1$ inherits from the object identified by $i_2$. In our intention, the object identified by $i_1$ is a dynamic role of the object identified by $i_2$. The relation $SS$ should not contain cycles and should be pure hierarchy, i.e. no role can be a property of two or more objects.

A role cannot exist alone, i.e. if the object identified by $i_2$ is deleted, then the object identified by $i_1$ is automatically deleted too. Roles of an object should be distinguished by some flag, but in our examples for simplicity we drop it considering self-evident. Note that we subdivide inheritance into \textit{static} and \textit{dynamic}, because the first kind of inheritance can be a compile-time property, while the second kind \textit{must be} a run-time property. Usually we assume that all identifiers of roles are among root objects identified by $R$.

\begin{figure}
\centering
\begin{tabular}{|l|}
\hline
\textbf{S – Objects (and roles)} \\
\hline
$<i_{13}, \text{Person}, \{ <i_2, \text{name}, \"Doe\">, <i_3, \text{born}, 1948 > \} >,$ \\
$<i_4, \text{Person}, \{ <i_5, \text{name}, \"Poe\">, <i_6, \text{born}, 1975 > \} >,$ \\
$<i_7, \text{Person}, \{ <i_8, \text{name}, \"Lee\">, <i_9, \text{born}, 1951 > \} >,$ \\
$<i_{13}, \text{Emp}, \{ <i_{14}, \text{sal}, 2500 >, <i_{15}, \text{worksIn}, i_{127} > \} >,$ \\
$<i_{16}, \text{Emp}, \{ <i_{17}, \text{sal}, 1500 >, <i_{18}, \text{worksIn}, i_{128} > \} >,$ \\
$<i_{19}, \text{Student}, \{ <i_{20}, \text{studentNo}, 223344 >, <i_{21}, \text{faculty}, \"Physics\" > \} >,$ \\
\hline
\textbf{C - Classes} \\
\hline
$<i_{40}, \text{PersonClass}, \{ <i_{41}, \text{age}, \ldots \text{code of the method Age...} >, \ldots \text{other properties of PersonClass...} \} >,$ \\
$<i_{50}, \text{EmpClass}, \{ <i_{51}, \text{changeSal}, \ldots \text{code of the method changeSal...} >, <i_{52}, \text{netSal}, \ldots \text{code of the method netSal...} >, \ldots \text{other properties of EmpClass...} \} >,$ \\
$<i_{60}, \text{StudentClass}, \{ <i_{61}, \text{avgScore}, \ldots \text{code of the method AvgScore...} >, \ldots \text{other properties of StudentClass...} \} >,$ \\
\hline
\textbf{R – Root identifiers} \\
\hline
$i_1, i_4, i_7, i_{13}, i_{16}, i_{19}, \ldots,$ \\
\hline
\textbf{CC - Inheritance between classes} \\
\hline
Empty. \\
\hline
\textbf{SC - Membership of objects and roles in classes} \\
\hline
<i_{13}, i_{40}>, <i_4, i_{40}>, <i_7, i_{40}>, <i_{13}, i_{50}>, <i_{16}, i_{50}>, <i_{19}, i_{60}>, \ldots,$ \\
\hline
\textbf{SS – Inheritance between roles and objects} \\
\hline
<i_{13}, i_4>, <i_{16}, i_7>, <i_{19}, i_7>, \ldots,$ \\
\hline
\end{tabular}
\caption{Example of an AS2 object store}
\end{figure}
In Fig.2.10 and Fig.2.11 we have omitted the static inheritance relation from EmpClass and StudentClass to PersonClass. It is substituted by the dynamic inheritance and object membership in classes. For instance, a Student role inherits dynamically properties of its Person object, hence indirectly imports properties of the PersonClass.

We will use the terminology in which each object consists of roles and one of them is the main role. For instance, a Lee’s object consists of three roles, Person, Emp and Student, and Person is its main role. Each object can be bound (retrieved) by using the name of any of its roles. However, the result of the binding will be a reference to the proper role (roles).

In our intention the AS2 model will have a natural delete and copy semantics. Deleting a role implies deleting all its sub-roles. In particular, deleting a main role means deleting of the whole object. Similarly for copying: we assume that copying a role implies automatic copying of all its sub-roles.

As for the AS1 model, the nature of dynamic inheritance will be explained later, when we define the query execution engine.

### 2.3.2 Peculiarities of the Object Model with Dynamic Object Roles

Below we list several features, which make the concept of dynamic roles different in comparison to the classical object-oriented concepts.

- **Substitutability.** The AS2 model gives up the substitutability principle. It is no more necessary. For instance, the typical statement that a Student object can be used in any place where the Person object can be used makes little sense for the AS2 model. An
object consists of many roles, each with own name, and the programmer makes an
explicit choice which role is proper in the given place of the program. If Lee has three
roles Person, Emp and Student, the programmer uses Person if he/she would like to
abstract from specialized roles, and uses Emp or Student in other program contexts which
require these roles. Of course, we can think on some mix of AS1 and AS2 models, where
substitutability coexists with dynamic roles.

• **Object identity.** An object has as many unique object identifiers as roles. One of the
identifiers (of the main role) is distinguished, but binding the object through a name of its
role returns an identifier of the role rather than this distinguished identifier.

• **Multiple inheritance.** Because roles are encapsulated there is no name conflict even if
the super classes would have different properties with the same name. There is no need
for EmployeeStudentClass, which inherits both from EmployeeClass and StudentClass.

• **Repeating inheritance.** An object can have two or more roles with the same name; for
instance, Brown can be an employee in two companies, with different Salary and Job.
Such a feature cannot be expressed by the traditional inheritance or multi-inheritance
concepts.

• **Multiple-aspect inheritance.** A class can be specialized according to many aspects. For
example, a vehicle can be specialized according to environment (ground, water, air)
and/or according to a drive (horse, motor, jet, etc.). Some modeling tools (e.g. UML)
cover this feature, but it is neglected in object-oriented programming and database
models. One-aspect inheritance makes problems with conceptual modeling and usually
requires multiple inheritance. Roles allow for avoiding problems with this feature.

• **Multiple interfaces.** Dynamic object roles in natural, universal and semantically
consistent way accomplish the idea of multiple interfaces to an object.

• **Variants** (unions). This feature, introduced e.g. in C++, CORBA and ODMG object
models, leads to a lot of semantic and implementation problems. Some professionals
argue that it is unnecessary, as it could be substituted by specialized classes. However, if
a given class can possess many properties with variants, then modeling this situation by
specialized classes leads to the combinatorial explosion of classes (e.g. for 5 properties
with binary variants - 32 specialized classes). Dynamic object roles avoid this problem.
Each branch of a variant can be considered a role of an object.

• **Object migration:** Roles may appear and disappear at run time without changing
identifiers of other roles. In terms of classical object models it means that an object can
change its classes without changing its identity. This feature can hardly be available in
classical object models, especially in models where binding objects is static.

• **Referential consistency:** In the presented model relationships are connected to roles, not
to the entire objects; thus, e.g. it is impossible to refer to Salary and Job of Smith when
one navigates to its object from the object School. In classical object-oriented models this
consistency is enforced by strong typing, but is problematic if the typing is semi-strong,
weak or absent.

• **Overriding:** Properties of a super-role can be overridden by properties of a sub-role. The
possibilities of overriding are extended in comparison to the classical object models: not
only methods but also attributes (with values) can be overridden.

• **Binding:** An object can be bound by the name of any of its roles, but the binding returns
the identifier of a role rather than the identifier of the object. By definition, the binding is
dynamic, because in a general case during compilation it is impossible to decide that a particular object has a role with a given name.

- **Typing**: A role must be associated with a name, because this is the only feature allowing the programmer to distinguish a role from another one. Hence, the role name is a property of its type (unlike classical programming languages, where a type usually does not determine the name of a corresponding object/variable). Because an object is seen through the names of its roles, it has as many types as it has different names for roles. Hence typing systems for the model with object roles must be redesigned.

- **Subtyping**: It can be defined as usual; for instance, the `Emp` type is defined with the use of the `Person` type. However, there is no sense to introduce the `StudentEmployee` type. Due to encapsulated roles, properties of a `Student` object and properties of an `Emp` object are not mixed up within a single structure.

- **Temporal properties**: Dynamic object roles are enormously useful for temporal databases, as roles can represent any past facts concerning objects, e.g. the employment history through many `Employee` roles within one `Person` object. Without roles, historical objects present a hard design problem, especially if one wants to avoid redundancy, to preserve reuse of unchanged properties through standard inheritance, and to avoid changing objects’ identifiers.

- **Aspects of objects and heterogeneous collections**: A big problem with classical database object models, for instance ODMG, is that an object belongs to at most one collection. This is contradictory with multiple inheritance, substitutability and open-close principle. For instance, we can include a `StudentEmp` object into the extent `Students`, but we cannot include it at the same time into the extent `Emps` (and vice versa). This may leads to inconsistent processing. Dynamic roles have the natural ability to model heterogeneous collections: an object is automatically included into as many collections as the types of roles it contains. For instance, in Fig.2.11 we can see three collections: `Person` (3 objects), `Emp` (2 objects) and `Student` (1 object) which in a specific way share information and overlap, but this does not lead to any conceptual problems. Note that if Lee will decide to study in parallel on two universities, his object will have two roles `Student`. Hence in the collection `Student` the Lee object will occur two times. This is logical and consistent, but such a situation is very hard to handle by the typical collection concept.

- **Aspect-Oriented Programming**: AOP makes it possible to encapsulate cross-cutting concerns within separate modules, for example, such concerns as: history of changes, security and privacy rules, visualization, synchronization, etc. As follows from the previous feature, dynamic object roles have conceptual similarities with AOP or can be considered as a technical facility supporting AOP.

- **Metadata support**: Metadata are a particular case of crosscutting concerns. Meta-information, as assumed e.g. in Dublin Core (such as authorship, validity, legal status, ownership, coding, etc.) can be implemented as dynamic roles of information objects.

As follows from the above, dynamic object roles have the potential to create new powerful qualities, which are difficult or impossible to achieve in the classical object model.
2.4 AS3 Store Model: Encapsulation and Information Hiding

Encapsulation is a human civilization principle which essence is abstraction from details of some artifacts, wrapping details into some non-transparent capsules, and manipulation of complex artifacts as closed atomic units. For instance, looking on the car we have a lot of encapsulated parts, for instance, a gear-box. Similarly in electronics: a chip encapsulates millions of details. As for other technical branches, encapsulation is also the basic principle of software engineering. It is applied during software production to reduce the complexity of software manufacturing processes and to reduce the complexity of software products. The reducing of the complexities is based on top-down or bottom-up strategies. The top-down strategy assumes that software producers design or manufacture more abstract entities, and then, subsequently rectify their details. In the bottom-up strategy details are wrapped into a more abstract whole which hide the details are not seen and the whole is manipulated as a black-box, only by defined external properties.

Programming languages introduce several concepts that address the encapsulation principle. Among them we have software modules, procedures, functions, abstract data types and classes. For instance, a procedure may encapsulate a lot of code which is not interesting for the programmer who uses calls of the procedure. He/she is interested only in what the procedure is doing and how it has to be invoked. Also the database domain has contributed to encapsulation by such concepts as relational tables (encapsulating complex physical implementation), database views, triggers, and perhaps others. The middleware technologies such as CORBA, Web Services, RMI and EJB, Enterprise Service Bus, virtual repositories, etc. also can be perceived as contributions to the encapsulation principle.

In this section we are interesting in the particular case of encapsulation that concerns objects and classes. While we consider encapsulation as inevitable, the problem is how it can be introduced to an object-oriented database model. Note that some particular way of introducing the concept can be the subject of critics. For instance, C.J.Date in the text entitled “Encapsulation is a Red Herring” strongly argues against the concept, claiming that it leads to impossibility to define query languages. Date’s arguments are correct; fortunately they do not concern encapsulation as such, but some popular variant of encapsulation that we do not accept too.

2.4.1 Orthogonal v/s Orthodox Encapsulation

There are several ways how encapsulation can be involved into the concepts of a module, class, object and type. The most popular way is known from C++; a similar idea is accomplished in Modula-2. The idea can be referred to as orthogonal encapsulation. In short, a module, a class, an object or a type has some internal properties which are assigned as public or private. (C++ and its successors have additionally some intermediate qualifier known as protected.) The assignment is fully in hands of the designer or programmer: he/she can assign the flag public or private to any property, independently of its kind. For instance, the flag public can be assigned to an attribute within an object, i.e. the orthogonal encapsulation does not exclude that some part of the object state will be seen outside the object or its class. As a consequence, the orthogonal encapsulation assumes that such visible parts of the object state are to be served by special generic language constructs, such as a generic assignment operator, generic delete operator, etc., which act directly on references to some elements of the object state. Obviously this idea of encapsulation makes no problem for the definition of query languages, because all attributes of an object are directly accessible. For instance, this kind of encapsulation is implicitly assumed in ODMG OQL.
Some methodologists presenting object-oriented notions worked out another encapsulation concept, which we refer to as orthodox encapsulation. According to this idea, the state of an object should be totally hidden for the programmers. All what one can do with an object should be determined by methods associated with it. Generic operations on the object state, such as assignments, inserts or deletes, are disallowed. In particular, generic operations on an attribute, e.g. Sal, are substituted by two methods: so-called getter (getSal), which returns the value of the attribute, and so-called setter (setSal), which changes the value of the attribute according to a parameter.

However, we have doubts concerning such an idea. We agree with the Date’s arguments, but the inference is different. We accept encapsulation but not the orthodox encapsulation. Even the Date’s argument that the (orthodox) encapsulation is contradictory to query languages is invalid and concerns syntax rather than semantics. If we assume that each getter has the same name as the name of the attribute that it gets, then it will be possible to construct a query language in the spirit of SQL, where all the attribute names would be in fact invocations of getters. This is simply a syntactic convention which is not difficult to accomplish in any system. Hence it is possible to build a query language, despite the orthodox encapsulation.

Our arguments against the orthodox encapsulation are much stronger:

• It is not true that the object-orientedness implies this kind of encapsulation. The mission of object-orientedness is seamless mapping between real-life business objects and computer-side data structures (called objects) and v/v. All other assertions concerning object-orientedness are rhetoric froth that we do not buy if it is not based on strong pragmatic considerations concerning usability, consistency and efficiency of programming interfaces.

• It is not true that direct access to an object state is error prone. For more than 40 years computer folks developed hundreds of useful programming and query languages that assume the direct access to attributes (of records, tuples, structures, etc.). The folks have written billions lines of useful code under this assumption. It didn’t result in any catastrophic disadvantage. The problem is artificially invented in shallow rhetoric of some object-oriented gurus.

• The orthodox encapsulation violates the object relativism principle. For instance, the programmer is allowed to use an assignment to an attribute on the level of a method code, but he/she is disallowed to do that on the level of an entire object. But an attribute is an object too and the principle claims for no syntactic and semantic differences concerning objects on any object hierarchy levels.

• The orthodox encapsulation violates the principle of orthogonality of collection types. For instance, an object can contain a repeating attribute Hobbies with unknown number of repetitions. Obviously, in such a case two methods, getHobbies and setHobbies, are not enough. In such case the object-oriented gurus recommend to use iterators, i.e. methods such as getFirstHobby, getNextHobby, isNextHobby?, etc. There are severe problems with iterators concerning their state, see [Bake93]. We put attention on another aspect. What would return a method such as getNextHobby? To achieve full universality (e.g. the possibility to remove a single hobby) the method should return a reference to this element of the object state. How such a reference has to be served by the language constructs? Again, the language designers should provide generic operations such as assignment and delete. Hence, we have received the contradiction: the orthodox encapsulation assumes that the object state is hidden, but iterators create a back door: they make a part of the state visible. The orthodox encapsulation assumes no generic
operations on elements of the object state, but iterators make little sense without such operations.

- Similar arguments as the above can be repeated in case when an object can contain null values, optional data, complex attributes or variants: in all such cases a getter and a setter is not enough to serve all the cases.

- The orthodox encapsulation is inconsistent with the idea of a database schema, which is an inevitable pragmatic component of any database programming interface.

- The orthodox encapsulation is contradictory to query languages if one assumes that an object can be complex, with optional, repeating, variant and complex attributes, can be linked by pointer links, can contain dynamic roles, etc.

- The orthodox encapsulation introduces conceptual difficulties with generic programming. For instance, it is possible that objects of different classes contain an attribute of some type that has to be served (updated) by external routines. The most convenient method is to pass a reference to such an attribute to the routines in the call-by-reference mode. The orthodox encapsulation makes this impossible.

Summing up, our final conclusion is the following:

For object-oriented databases the orthodox encapsulation (in which the state of an object is totally hidden) is a conceptual nonsense.

In our proposal we follow the orthogonal encapsulation, where each property of a module, class, object or type can be public or private, independently of its kind.

### 2.4.2 Formal Basis of the AS3 Store Model

In the AS3 model we augment a class of the AS1 or AS2 model with an export list containing the names of all public properties of the class and of the objects being the members of this class.

By default, if an export list is not defined for a particular class it means that all properties of the class and members of this class are public. Public properties are available externally according to regular scoping rules (that we introduce later). Private properties (not present on a corresponding export list) are available only within bodies of the methods that are stored within the corresponding class. We can easily introduce another privacy kind known as protected; we discuss this small issue later, when we present the semantics of classes and scoping rules for binding names.

<table>
<thead>
<tr>
<th>S – Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; i₁, Person, { &lt; i₂, name, &quot;Doe&quot;&gt;, ... } &gt;,</td>
</tr>
<tr>
<td>&lt; i₅, Emp, { &lt; i₆, name, &quot;Poe&quot;&gt;, &lt; i₇, sal, 2000&gt;, &lt; i₈, worksIn, i₂₂&gt;, ... } &gt;,</td>
</tr>
<tr>
<td>&lt; i₉, Emp, { &lt; i₁₀, name, &quot;Lee&quot;&gt;, &lt; i₁₁, sal, 900&gt;, &lt; i₁₆, worksIn, i₃₃&gt;, ...} &gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C - Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; i₄₀, PersonClass, { &lt; i₄₁, age, (...the code of the method age)&gt;, ... other invariants of the PersonClass... } &gt;,</td>
</tr>
<tr>
<td>&lt; i₅₀, EmpClass, { &lt; i₅₁, changeSal, (... the code of the method changeSal...)&gt;,</td>
</tr>
<tr>
<td>&lt; i₅₂, netSal, (... the code of the method netSal ...)&gt;},</td>
</tr>
<tr>
<td>&lt; i₅₃, ExportList, (changeSal, netSal, worksIn) &gt;,</td>
</tr>
</tbody>
</table>
... other invariants of the EmpClass... } >

R - Start identifiers

\( i_1, i_5, i_9 \)

CC - Inheritance among classes

\( < i_{50}, i_{40} > \)

SC - Membership of objects within classes

\( < i_1, i_{40} >, < i_5, i_{50} >, < i_9, i_{50} > \)

Fig. 2.12. Example of an AS3 object store extending an AS1 object store by export lists

The name ExportList is reserved for the internal use only. By this declaration we assume that all properties of PersonClass and Person objects are public. Concerning the EmpClass and Emp objects, properties named changeSal, netSal, worksIn are public, the property sal and other properties are private, hence available only in bodies of the methods changeSal and netSal. So far we present only situation in the object store. Detailed discussion concerning how export lists will be handled by the semantics of a query language will be presented later.
3 Environment Stack, Query Results and Function

The concept of \textit{environment stack} (aka \textit{environmental stack} or \textit{call stack}) appeared in 50-ties or early 60-ties of the previous century, together with compilers of oldest high-level programming languages such as Algol-60. From that time the stack is a basic concept of semantics and implementation of majority of well-known languages, including Pascal, C, C++, Smalltalk, Java, C#, etc. The idea of the stack is well-known for the developers of compilers. It is simple and obvious, but not always well explained in manuals and textbooks. According to our experience, the database community folks rarely realize what the stack is for, what problems it solves and which properties it has. It is quite common confusing it with another stack known as \textit{arithmetic stack} or \textit{result stack} which is necessary for quite different reasons. Lack of proper knowledge, simplistic reasoning, neglecting hard semantic problems and hurrah-theoretical attitude are probably the reasons that developers of popular theories of query languages (such as relational algebra, relational calculus and formal logic) totally ignore this concept, or consider it an implementation issue having no relevance to high-level semantics. Unfortunately for the world (and fortunately for us) such theories are dramatically limited w.r.t. to a lot of semantic phenomena that occur in query languages such as SQL. Moreover, richer database models, such us object-oriented and XML-oriented models, show inadequacy of these theories. Even simplest queries, such as $2+2$ or \textbf{select} \textit{Salary} - \textit{Tax} \textbf{from} \textit{Employees} are non-expressible in these theories. However real problems with these theories appear when one would like to describe unified semantics for languages that integrate querying and programming. Nowadays there are many such languages, including recent SQL standards, Oracle PL/SQL, MS Transact SQL, some 4GLs, C#/LINQ, and others. Therefore we abandon these theories as worthless for our purposes. Instead of them we introduce concepts with no such limitations. We will show that all advantages of these theories over our concepts, including their potential for query optimization, distributed query processing, etc., are overrated and inessential. We do that in a way that is better, more general and corresponding to real practical query languages.

In this way we are coming back to the idea of an environment stack. We will consider its roles in programming languages to conclude that the same reasons for introducing it occur in query languages. To this end, however, we will need to modify the stack structure to accomplish new features stemming from the properties of query languages.

3.1 Environment Stack in Programming Languages

The concept of \textit{environment} of program execution denotes all the run-time entities (variables, constants, objects, functions, procedures, types, classes, etc.) that are available at the given point of the program control. The environment is not a flat structure and is changing during the run of the program. The most convenient (and clear for the programmers) assumption is that that the environment is subdivided into sub-environments that appear and disappear during the program run. From the ergonomic point of view there is a need to follow some principles that govern the appearance and disappearance of the sub-environments, as follows:

\begin{itemize}
  \item Local sub-environment of the given programming abstraction (e.g. a procedure) has the priority over more global sub-environments. For instance, the programmer working on a procedure focuses on its local environment and only sometimes refers to more global ones, e.g. to global (static) variables.
\end{itemize}
The principle of a local context (lexical scoping): the programmer developing a procedure is unable to take into account such elements of sub-environments that are unknown for him/her during the time of writing this procedure. Hence such sub-environments should be invisible during the run time, to avoid false binding (unexpected binding to a random element, e.g. due to a mistake in a variable name).

The principle of free nesting of procedure calls: the programmer working on a procedure can freely call within it other procedures. He/she takes into account their parameters, but disregards all other elements of their sub-environments (e.g. their local variables). The number of nested calls is unlimited, recursive calls should be allowed and they should not imply anomalies, inconsistencies or the need of special treatment.

These principles lead to the concept of environment stack, which is just responsible for appearance and disappearance of sub-environments related to particular procedures. The stack fulfills the following roles (in our terminology functions and methods are procedures too):

- Control over scopes of variable names occurring in a program and binding these names to run-time entities;
- Storing values of local variables of procedures;
- Storing values of actual parameters of procedures;
- Storing a return track, i.e. an address of the program code where the control should be returned after the given procedure is terminated (usually the next address after the procedure call).

A stack is a main memory (or virtual memory) data structure and is assigned to a single client application program or to a single process or thread. It is subdivided into sections, where each section (also called activation record) corresponds to a particular sub-environment. A section is associated with a particular procedure call or an executed program block. When the control is shifted to a procedure call, a new section with all entities local to this call is pushed on the top of the stack. The same is done when the control is shifted to a module (except that a return track is unnecessary). The section is popped from the stack when the procedure or program block is terminated. For the procedure that is currently running all values of parameters, local variables/objects and any other local entities are stored within the top stack section.

Note that the environment stack has more functionality than a typical stack understood as an abstract data type with four operators: push (push a new element on the top), pop (remove an element from the top), top (read top element) and isEmpty? (checking if the stack is empty). Additional functionalities concern name binding (which implies the search on the entire stack), scoping rule (skipping visiting some sections) and (in some cases) inserting new sections in the middle of the stack. All these additional functionalities stem from the definition of semantics and will be explained later.

3.2 Name Binding

Binding is a compiler or run-time action that for a given name occurring in a source program subordinates a corresponding run-time entity, e.g. a main memory address, an object identifier, a start addresses of an executable procedure code, etc.

For instance, binding of a variable $X$ is a substitution of this name by a main-memory address, where the value of $X$ will be recorded. All occurrences of $X$ in a source code are substituted by this address.
Providing an environment stack stores environments together with all source names of runtime entities, a binding action for some name X is performed according to the following steps:

- The machine checks the top of the stack for an entity named X; if there is such an entity, the binding returns it as the result and the action is terminated.
- If the top does not contain an entity named X, a section below the top is checked.
- Such a process is continued in lower and lower stack sections, till the entity named X is found. Visiting particular stack sections is governed by scoping rules that require omitting some sections;
- If X is not found on the stack, then the global environment is searched. The global environment contains static variables, database objects, computer environment variables, procedure libraries, etc. Alternatively, we can assume that the global environment is the lowest stack section - in such a case X must be found on the stack, otherwise an error should be reported.

In fact, this search is usually done during compile time on the static environment stack rather than on run-time stack; we will return to this issue a bit later. In Fig.3.1 we show the case when name X occurs within a block b that is a part of procedure p2 that is called from procedure p1. The control is within the block b. Arrows show the order of searching the stack. The search is terminated when X is found.

In Fig.3.1 we also show the scoping rule: variables and actual parameters of the procedure p1, and perhaps former sections of procedures, are not seen. The search is going directly to global variables. Procedures p1 and p2 can be written at different time, by different programmers who do not and need not to communicate and make some agreement. The programmer writing procedure p1 can use the name X and the programmer writing procedure p2 can use X, but these two X denote different things. Due to scoping rules X occurring within p1 is not seen when p2 is processed, thus there is no conflict.

If a programming language introduces modules, classes, inheritance and perhaps other notions the situation on the environment stack and scoping rules can be much more complex. We will discuss this issue after we introduce AS0-AS3 store models.
The stack mechanism makes it possible to accomplish the following features of programming languages:

- **Abstraction and encapsulation**: the inside of a written procedure is hidden from the programmers that use it. The procedure is seen only by its interface consisting of its name, names and types of parameters the type of its output, and the informal description of consequences of a procedure action.

- **Isolation**: the programmer writing different procedures need not know about each other and need not to make some agreements concerning names of properties that are involved inside the procedures.

- **Semantic independency and reuse**: a procedure with a well specified interface and performance can be invoked from many places of an applications or different applications.

- **Unlimited invocations of procedures from other procedures**: Because a stack section is assigned to a procedure call (rather than to a procedure body) there is no conflict concerning procedures’ local environments. Recursive calls are also possible, providing the memory for the environment stack is sufficient and recursive invocations are somehow terminated within the procedure body.

- **Consistent management of names used in programs**: The name spaces are assigned to particular procedure calls and there in no naming conflicts between local procedures’ environment.

- **Implementation of parameter passing methods**: values of parameters and other their properties are recorded within stack sections thus the consistent access to the parameters is possible. The environment stack can accomplish various parameter passing methods, such as call-by-value, call-by-reference, strict-call-by-value, and perhaps others. (We consider parameter passing methods later.)

As we will show, the above features have fundamental meaning for query languages, allowing one to accomplish their properties in a simple, consistent and universal way. This concerns, in particular, the possibility to define auxiliary names (“correlation variables”) that are defined within queries, the possibility of nesting queries, scope rules for all names occurring in queries, combining in queries names of database properties with names of applications’ properties, and so on. The stack mechanism is absolutely inevitable if one seriously thinks on integration of queries with programming constructs and abstractions. Lack of the environment stack concept in artifacts such as various algebras (object algebras, in particular), F-logic, Datalog, monoid calculus, OQL, XQuery, etc. in advance sentences them to limitations and handicaps.

### 3.3 Static and Dynamic Environment Stack

In languages with *early (compile-time) binding* names occurring in a program have the second-class citizenship, i.e. they are lost after compilation. The names do not exist during run-time. For such languages the environment stack must exists in two forms: a *static stack* that is managed during compilation, and a *dynamic stack* managed during run-time. The binding of source program names is performed on the static stack. During compilation this stack simulates the behavior of the dynamic stack, i.e. it grows and shrinks according to the moving of the syntactic analysis control point along the source program being compiled. In such cases binding is subdivided into two phases. In the static phase binding of variables is
expressed relatively to the size of stack, i.e. for each variable name the static binding returns so-called offset, e.g. the distance in bytes between the top of the dynamic stack and the address of the run-time entity corresponding to this name. Eventually the binding is performed during run-time, when the address of the stack top is known. For instance, the address of the variable name is calculated by subtracting the offset from the address of the stack top.

Due to some irregularities of data structures practically all languages work with more advanced dynamic binding, when variable names (in some representation) still exist on the dynamic stack. In interpreted (script) languages the binding is fully dynamic, i.e. names of variables, together with their values, physically exist on the dynamic stack. The subdivision between static binding (during compile time) and dynamic binding (during run time) is almost always a bit rough and implementation dependent: usually the static binding is supported by calculations during run time, and dynamic binding operations (for optimization) are partly done during compile time.

Query languages are interpreted hence for them the dynamic binding and dynamic stack is relevant. To avoid considerations related to physical implementation of the static and dynamic stacks we assume that all bindings are dynamic. Hence at least at the beginning we avoid the static stack. All run-time entities that are necessary for query processing will be stored at the run-time stack together with their source names. In fact, majority of the semantics of query languages we are able to explain without referring to the static environment stack.

However, some issues in query languages require the static stack. These are the following:

- **Strong static type checking**: obviously this feature requires the static environment stack, because strong static type checking of queries and programs is a compile-time process. This issue we discuss and explain much later, when we introduce all the concepts necessary for the definition of semantics.

- **Query optimization**: almost all query optimization methods, including factoring out independent sub-queries, removing dead sub-queries, methods based on indices, etc. require operations on the static environment stack. Optimization may also concern binding actions, which can be partly resolved due to the static stack. As previously, we will discuss and explain the feature much later.

- **Resolving some ambiguities in queries**: for instance, elliptic queries or queries acting on irregular (semi-structured) data require actions on the static environment stack e.g. to avoid false bindings.

In our further discussion we present the stack in conceptual rather than physical form. In this way our considerations will be still abstract and high-level. Of course, physical implementation of the stack can much differ in details from our conceptual view.

### 3.4 Environment Stack in the AS0 Store Model

In our presentation we focus on the conceptual view of the environment stack rather than its physical implementation. The construction of the stack will reflect basic concepts of the query languages’ semantics that we have introduced in the AS0 store model. In the following, operations on the stack will be explained for AS1, AS2 and AS3 models, but essentially the construction of the stack for these models remains the same. In contrast to typical programming languages the stack will be prepared to treat uniformly individual data and collections. The stack is a client-side main-memory structure; hence its size is somehow limited. However, this is not a significant limitation, because nowadays main memories are
very large and can be extended by virtual memories. For instance, in Loqis all main memory structures, including stacks and indices, can be partly stored on a disc, depending on some policy involving the statistical frequency of the use. Moreover, there are a lot of possible optimizations aiming reduction of the stack size. Thus the danger that the main-memory stack implies some limits in scalability of applications in our opinion does not occur. Actually, we see no reason why e.g. SQL is scalable and query languages based on SBA are not scalable (as claimed by some authors); such opinions are based on immature superficial reasoning rather than on deep analysis of technical properties. Because the stack is a client-side structure, its idea is orthogonal to geographical distribution of database resources. Hence the argument of some authors that the stack implies no possibility of processing distributed databases has no foundation at all (what we have also confirmed by implementation of distributed tools and applications).

The environment stack in SBA is abbreviated ENVS. It consists of sections, which are ordered. The newest section will be named the top and the oldest one will be named the bottom. As in case of programming languages, each section presents information on some run-time environment, for instance, a local environment of a procedure, a local environment of an object, of a class, of a database, etc. The size of a section is conceptually unlimited.

The bottom of ENVS will contain global sections, in particular, a database section, a section related to the current user session, libraries that are available in the computer environment and some computer environment variables, such as date, time, user login information, etc. We assume that each name that may occur in source query must possess its counterpart on ENVS, hence the stack will be the only name binding mechanism; no name occurring in a query can be bound otherwise. For simplicity, in examples we sometimes consider only the database section, neglecting other sections. In real implementations these sections must be of course taken into account.

ENVS will be prepared to store uniformly information on persistent and volatile objects. Persistent objects are usually shared among many user sessions. Volatile objects are properties of a single user session and are not available for other sessions. Local objects of procedures are also considered volatile.

ENVS will also be prepared to store all information that stems from definitions that are local for queries. Almost all query languages introduce such definitions; for instance, “correlation variables” in SQL, names defined by as operator in OQL and for operator in XQuery, names bound by quantifiers, cursors in for each statements, etc. All such names occurring in queries will be bound according to the same standard routine.

In programming languages the run-time environment stack usually plays also the role of the store of values of all volatile entities (e.g. local variables of procedures). In contrast, we assume that the stack and the object store are separate notions. We have several reasons to separate the notions. The main reason is conceptual - the separation is quite convenient for the definition of the semantics of a query language. The second reason is that we consider persistent objects which live on a database server machine, while the stack lives on a client machine. Hence the stack may contain references to persistent objects, but not the objects themselves. The third reason is that the same object can be referred from several stack sections, thus it is impossible to include it to a single section. In implementation it is possible to make some optimizations which assume that some volatile objects live on the stack. Such optimization was implemented in Loqis. However, in this presentation we do not deal with implementation issues.
3.4.1 The Concept of Binder

A basic structure that is stored at ENVS is called **binder**. The concept is specific only for SBA, it does not occur in other approaches to query languages. Binder is also a key concept for query languages defined through SBA. It is incredible how many semantic issues can be explained through such a simple notion. It is also incredible that it was not introduced in any previous theory devoted to query languages. Binder supports universal solutions to all semantic issues related to naming, scoping, binding, parameter passing, and definition of many query operators. Lack of this concept in former theories, (such as algebras and calculi) is caused by the fact that naming issues have the second-class citizenship in these theories: names are properties of informal meta-languages rather than the defined languages. In contrast, naming issues are fundamental in all known query languages, including SQL, OQL and XQuery, hence lack of formal treatment of names inevitably results in imprecise, underspecified semantics.

Binder is a pair $n(x)$, where $n \in N$ is any external name which can be written in a source query or program, and $x$ is any result of a query, in particular, a reference to an object or a value.

Results of queries (i.e. the domain Result) will be defined a bit later. For better legibility the binder is written $n(x)$ rather than <$n$, $x$>. The idea of a binder is very simple. Its role is to binding names, hence it joins an external name $n$ with a run-time entity $x$ that is denoted by this name. Binding name $n$ means that we are looking at ENVS for a binder (binders) $n(x)$. The result of the binding is $x$. If the stack contains no such a binder, the situation can be qualified as a binding error. Alternatively, for semi-structured data the situation can be qualified as correct, but the result of the binding is the empty bag. However, we reject such an alternative, because it makes impossible to distinguish absent data (normal in semi-structured data) from spelling mistakes in names. We return to this issue much later, when we will consider processing semi-structured data in SBA.

In our semantics binders will be present not only on the environment stack. Some query operators will return binders as query results. However, eventually the mission of every binder is to appear sooner or later on the environment stack.

3.4.2 Definition of an Environment Stack

In SBA the environment stack ENVS is a sequence of sections, where each section is a set of binders. The sections correspond to particular run-time environments.

Fig.3.2 presents a sample state of the object store (see Fig.2.1 and Fig.2.2) and Fig.3.3 presents a sample state of the environment stack that address this store. In this example binders contain object identifiers (references) and values. Single references and single values are particular cases of query results, but in general, because query results can be nested and arbitrarily complex, binders can be nested and complex too.
Note that the stack mixes and unifies volatile and persistent data. This is due to the orthogonal persistence principle which requires no difference between querying and operations on persistent and volatile data.

We remind that in SBA the concept of State consists of the state of an object store, as shown in Fig.3.2, and the state of the environment stack, as shown in Fig.3.3. Lack of formal and precise definition of the concept of state (the state of an object store and the state of an environment stack) is one of the reasons that the semantics of query language standards, such as

<table>
<thead>
<tr>
<th>Temporary processing section</th>
<th>Top</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volatile (non-shared) objects</td>
<td>Emp(i1)</td>
</tr>
<tr>
<td>Persistent (shared) objects</td>
<td>X(i_{127}) Y(i_{128})</td>
</tr>
<tr>
<td>- properties of an executed procedure</td>
<td>N(5) I(&quot;Maria&quot;)</td>
</tr>
<tr>
<td>- properties of a processed object</td>
<td>..........</td>
</tr>
<tr>
<td>- binders to volatile properties of the current client session</td>
<td>name(i_{10}) sal(i_{11}) address(i_{12}) worksIn(i_{16})</td>
</tr>
<tr>
<td>- binders to global library functions</td>
<td>..........</td>
</tr>
<tr>
<td>- binders to properties of the computer environment</td>
<td>Emp(i_1) Emp(i_5) Emp(i_9) ... Dept(i_{17}) Dept(i_{22}) ...</td>
</tr>
</tbody>
</table>
SQL-99, OQL and XQuery, is imprecise and leading to a lot of incompatible decisions in different implementations.

We remind that an environment stack is a volatile data structure that is maintained at the side of a client (a user session). In case of concurrent access of many clients, each of them maintains an own stack. If at some time there is no client session, then there is no environment stack. Moreover, if a client session has some parallel processes, threads or transaction, then it is possible that there are as many stacks as processes/threads/transaction within the session. When a session (process/thread/transaction) is started a stack dedicated to it is created and fulfilled in advance by global sections, as shown in Fig.3.3. Then the stack is maintained by the query execution engine, as will be determined in the following parts of this SBA description.

Relational systems based on SQL assume that all the query processing is performed on the side of a database server rather than on a client. This architecture has some advantages, because source SQL queries (essentially, strings) and query outputs are the only units that circulate among clients and the server. The server centralizes all the mechanisms of query optimization and processing. The architecture has also severe disadvantage if there are many (hundreds or thousands) of clients, because query processing could become a bottleneck compromising the client-side performance. Thus we will strive to develop an architecture in which clients are responsible for query optimization and processing as much as possible, reducing the server workload. However, this does not mean that the environment stack concept enforces such architecture. If all the query processing has to be done on the server, then each transaction on the side of the server should maintain an own environment stack, as described above.

3.4.3 ENVS and Name Binding

As we have presented before, semantics of a query is a function which maps a state into a query result. Because in SBA each single name is a legal query, the question is how the mapping looks for it. This mapping we call binding. Binding is performed on ENVS according to the very simple “search from the top” rule. As we have argued in the part devoted to the environment stack in programming languages, some sections must be skipped because of scoping rules; so far we abstract from this feature. Fig.3.4 presents the order of searching within ENVS. Binding name $n$ requires searching for the binder $n(x)$ which is present on the stack, according to the rule. The first success stops the search. The result of the binding is $x$. To take into account collections we assume multi-valued binding. If the binding of a name $n$ succeeds in some section, and it contains more binders named $n$: $n(x_1), n(x_2), \ldots, n(x_k)$, then all such binders are taken into account and the result of the binding is the bag\{ $x_1, x_2, \ldots, x_k$\}.
The function that does the search we call `bind`. It has a name as an argument and returns a value being the result of the binding. The result will be stored at another query result stack that will be explained later. The function is quite easy to explain, for instance (Fig.3.4):

\[
\begin{align*}
\text{bind}(X) &= i_{127} \\
\text{bind}(I) &= \text{"Maria"} \\
\text{bind}(\text{sal}) &= i_{11} \\
\text{bind}(\text{Dept}) &= \text{bag}\{i_{17}, i_{22}, \ldots\} \\
\text{bind}(\text{Emp}) &= i_1
\end{align*}
\]

Note that in the last case there is a section that contains more binders named `Emp`; however, because the top section contains a single binder named `Emp`, the search is terminated on this binder. This feature is known as overriding; it has significance in query languages that we intend to cover and more generally, in object-oriented query/programming languages. The overriding accomplishes the principle of a local context (lexical scoping), which provides that a particular name is bound in the environment that is currently most close according to the programmer thinking. Note also that the binder `Emp(i_1)` occurs two times on the stack. Such a case is normal in query languages that we want to cover.

### 3.5 Results Returned by Queries

Now we are prepared to define the domain `Result` of queries, as explained previously. As we noted before, in SBA we assume that queries never return objects but references to objects, sometimes within more complex structures. Objects live in the object store; no entity called object occurs elsewhere. Queries can also return values stored in objects and values calculated by some functions or algorithms. As in other approaches, we introduce structures, bags and sequences as results of queries. An essential novelty of SBA in comparison to other theories and proposals of query languages is that queries can return named structures that we already
know as *binders*. Binders, as a structures returned by queries allow us to deal with naming issues that occur in the semantics of query languages.

Generalization of the above assumptions leads us to the following recursive definition (c.f. the AS0 store model):

- Any atomic value belonging to \( V \), or a value calculated by query operators, belongs to *Result*.
- Each reference to an object (object identifier) belonging to \( I \) belongs to *Result*. In particular, the domain *Result* includes also references to procedures, functions, views, methods and other behavioral entities if they will be introduced in a particular model of the object store.
- If \( x \in \text{Result} \) and \( n \) is an external name, \( n \in N \), then the binder \( n(x) \) belongs to *Result*. Such a result we will also call *named value*.
- If \( x_1, x_2, x_3, \ldots \) belong to *Result*, then \[ \text{struct}\{ x_1, x_2, x_3, \ldots \} \] belongs to *Result*. Token **struct** is a structure constructor, which can be implemented as a flag decorating a result. The order of elements in a structure can be significant. In contrast to typical structures known from e.g. C/C++, we do not assume that all elements of structures must be named (i.e. in our terminology, elements need not to be binders). We assume full freedom in this respect, which is constrained only by assumed query operators and typing (which will be introduced later). In particular, we assume that some or all elements of structures can be unnamed, and we do not exclude the situation when two or more elements of a structure have a same name. A structure will be considered as a single (but composite) element, i.e. a structure is not a collection. As usual, types of \( x_1, x_2, x_3, \ldots \) can be different. Implicitly, we assume that if a structure has a single element, \[ \text{struct}\{ x_1 \} \], then it is equivalent to this element \( x_1 \). Structures having no element are not allowed. Our structure generalizes the concept of tuple, as known from relational systems.
- If \( x_1, x_2, x_3, \ldots \) belong to *Result*, then \[ \text{bag}\{ x_1, x_2, x_3, \ldots \} \] belongs to *Result* and \[ \text{sequence}\{ x_1, x_2, x_3, \ldots \} \] belongs to *Result*. **bag** and **sequence** are collection constructors, which can be implemented similarly as **struct**, by a flag decorating a query result.
- The set *Result* has no other elements.

Note that **struct**, **bag** and **sequence** are *constructors of values* rather than constructors of types. We underline names of these constructors to avoid confusing them with keywords **struct**, **bag** and **sequence** that will be introduced in SBQL. The definitions of **struct**, **bag** and **sequence** constructors are unified, what supports uniformity and compact definition of semantics. Note however that:

\[
\text{struct}\{ x_1, x_2, x_3, \ldots \} \text{ is considered an individual element (it is not a collection), while } \text{bag}\{ x_1, x_2, x_3, \ldots \} \text{ and } \text{sequence}\{ x_1, x_2, x_3, \ldots \} \text{ are collections.}
\]

Although in the AS0 store model we do not introduce sequences, some operators can make sequences from bags (e.g. the *order by* operator); hence query results possess such an option. We do not introduce other collections, for instance *sets*, because we consider them particular cases of the two introduced collections. If one will consider the necessity of the set collection concept in some query language, it can be introduced on the level of types (rather than on the level of query results), providing automatic coercions between types which enforce checking for no duplicates and/or removing duplicates. In this model we do not assume that elements of collections must be of the same type. So far we do not determine what actually is a type, thus
such assertions might express some informal intention rather than a formal constraint. The concept of type is more complex than one can expect, thus we return to this issue much later.

Names used within named results (binders) are not constrained to names from the object store - they can determined ad hoc by the programmer through operators as and group as (similarly to auxiliary names in OQL). Moreover, if a query result is a binder n(i), where i is a reference to an object <i, m, ...>, then it is possible that n ≠ m, i.e. name of the binder is different from the name of an object. This property is obviously necessary for query languages; however it becomes difficult to introduce assuming classical typing systems. Hence the typing system for SBA must be significantly altered in comparison to classical ones; we discuss and introduce all such typing issues later. Below we present several examples of query results.

| Atomic: | 25  
| "Maria"  
| true  |
| Identifiers: | i1  
| i127  |
| Binders: | Dept(i22)  
| d(i22)  
| sal(i11)  
| counter( 5 )  |
| Complex: | struct{ i1, i17 }  
| sequence{ i1, i5, i9 }  
| bag{ struct{ i1, i17 }, struct{ i5, i22 }, struct{ i9, i22 } }  
| bag{ struct{ n("Doe"), sal(2500), d(i22) } }  
| bag{ struct{ Dept(i12), Emp( bag{ struct{ n("Doe"), s(2500) } } ), struct{ Dept(i22), Emp( bag{ struct{ n("Poe"), s(2000) }, struct{ n("Lee"), s(900) } } ) } }  

As seen from the above examples, we can create results which much remain objects. However our results never have their own identifier (identity) and they cannot be directly members of classes. However, they may contain identifiers of objects, which in turn can be members of classes. This will be explained in detail during discussion concerning the AS1-AS3 store models.

To make some association with relational query languages some query results can be represented as tables. This representation may simplify some examples by more legible visual representation. In Fig.3.5 we show how some query results can be depicted in this way. Such a representation may also have some meaning in implementation, as an optimization method. However, it has no meaning for formal semantics of query languages. Note that our query results cover the nested relational model (see last examples) and much more.
The presented above definition of query results is not dedicated to any specific query language. It is a sufficiently abstract and universal, thus can be adopted by any of them, including SQL, OQL and XQuery (providing their semantics will be expressed in terms of SBA). This definition will be the basis for the Stack-Based Query Language (SBQL) - a query language based on abstract syntax and the SBA method of specification of semantics.

### 3.6 Query Result Stack (QRES)

The query result stack (QRES) is necessary to accumulate temporary and final query results. As in the case of ENVS, the stack is a client-side main memory data structure. In contrast to ENVS, QRES is not a component of the State, because the result of query evaluation does not depend on the state of this stack. QRES is purely abstract data type, that is, it can be managed by four abstract operations:

- **void**: `push( newSection: queryResultType )` - push a new section above the top,
- **void**: `pop( )` - remove the top,
- `queryResultType: top( )` - read the top section,
- **boolean**: `isEmpty( )` - check if the stack is empty.

Each QRES section contains a query result, as defined in the previous subsection. Thus QRES is a straightforward generalization of an *arithmetic stack* known from implementation of programming languages and construction of computer hardware. Fig. 3.6 presents subsequent states of the arithmetic stack during calculation of a simple expression. At the beginning the stack is empty. The final state contains the result of the expression.

```
sequence( i1, i5, i11 )
  ↓
  ↓
  ↓
i1  i5
  ↓
i11

sequence( i1, i56, i72 )
  ↓
  ↓
  ↓
i1  i56  i72
  ↓
i11

<table>
<thead>
<tr>
<th>n</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Doe&quot;</td>
<td>i9</td>
</tr>
<tr>
<td>&quot;Poë&quot;</td>
<td>i14</td>
</tr>
<tr>
<td>&quot;Lee&quot;</td>
<td>i38</td>
</tr>
</tbody>
</table>
```
In implementation QRES can be also used for other purposes, for instance, to keep counters of iteration loops during processing collections.

QRES is not a necessary concept of a programming or query languages’ semantics. In some formulations, e.g. in the denotational semantics or expressing the semantics by recursive procedures, it disappears. However, in the operational semantics it is a very useful concept, as it allows one to explain precisely what is actually done within the abstract implementation machine. The advantages of the explicit QRES stack concept are the following:

- It makes the definition of semantics highly intuitive. Our experience with explaining the denotational semantics for average developers and programmers is very negative. We did not discover any essential advantage of this specification style.

- The stack concept makes it possible to reduce the distance between abstract specification and concrete implementation. It is a useful element of the abstract implementation paradigm that we follow in our presentation of SBA.

- The stack is associated with its static counterpart (S_QRES - it will be introduced much later) that is managed during query compilation. The static result stack is an inevitable feature of strong static type checking and query optimization methods.

Operations on QRES affect only its top, other stack sections are invisible. A section becomes visible after popping a proper number of top sections. Fig.3.7 presents an example state of the stack in SBA.

After evaluation of a query the top of QRES contains the final query result. This result is consumed by some agent within the application software, e.g. by a print command, by a
graphical user interface, by an updating or deleting clause, etc. Providing the application is free of inconsistencies, the beginning state of QRES is empty and the final state of QRES is empty too (all results of queries are consumed by the application software).

### 3.7 Opening a New Section of ENVS

In the following we assume that the terms environment, name scope and stack section mean essentially the same. In classical programming languages opening a new scope on the environment stack is caused by entering a new procedure (function, method) or entering a new block. Respectively, removing the scope is performed when the control leaves the body of the procedure or the body of the block.

To these classical situations of opening a new scope on the environment stack we add a new one. It is the essence and motive of SBA. The idea is that some query operators (called non-algebraic) behave on the stack similarly to program blocks. For instance, in the SBQL query:

\[
\text{Emp where ( name = "Poe" and sal > 1000 )}
\]

the part \((\text{name} = \text{"Poe" and sal > 1000})\) behaves as a program block executed in an environment consisting of the interior of an Emp object. Because we have assumed that each name occurring in a query has to be bound on the stack, this concerns also names name and sal. Hence, to make the binding possible we push on ENVS a section with the interior of the currently processed Emp object. The situation is illustrated in Fig.3.8 (c.f. Fig.2.2). The operator where iterates over the result of the query Emp. In each cycle it pushes onto the ENVS a section with the internal environment of a next Emp object. After processing of the condition the section is removed from the stack. Note that the interior of an Emp object is represented on the stack as a set of binders leading to properties of the object.

![Fig.3.8. Pushing a new section on ENVS to evaluate a condition](image)

Although the idea seems to be very simple, we will show that after generalization it is quite universal for many classical and non-classical query operators that occur in SQL, OQL and other query languages. The idea makes it possible to make precise, formal and general specification of the semantics of these operators. Moreover, the idea makes it possible to accommodate in query languages all the features and principles of the object-orientedness,
such as inheritance, polymorphism and encapsulation. The idea greatly supports uniformity, orthogonality and non-redundancy of query languages’ constructs. This will be shown on the SBQL case. We emphasize once again that the situation presented on ENVS is an abstract conceptual view that is necessary to define the semantics. In implementation the construction of the stack and operations on the stack can be significantly optimized.

For the AS1, AS2 and AS3 store models the basic construction of the environment stack is the same, but its behavior is different. This will be explained in proper sections devoted to semantics of SBQL for these models.

3.8 Function nested

In the previous part we have introduced the concept of interior of an object, which is to be pushed on the top of ENVS. Now we formalize the concept. In the formalization we strive to generalize it to cover all the cases that may require pushing on the stack some new environment. The formalization is quite simple and it is expressed through the function named nested. The function takes any query result (as defined previously) as an argument and is implicitly parameterized by an object store. For the argument it returns a set of binders. This set is assumed to be pushed at the top of ENVS, but the function nested does not do that. Pushing the result of the function nested on ENVS will be the subject of other mechanisms of the query execution engine. The function nested is defined as follows:

- For a complex object \(<i, n, \{ <i_1, n_1, ... >, <i_2, n_2, ... >, ..., <i_k, n_k, ... > \}>\) it holds:
  \(\text{nested}(i) = \{ n_1(i_1), n_2(i_2), ..., n_k(i_k) \}\). The case is illustrated in Fig.3.9. Indeed, \(\text{nested}\) returns the interior of the object identified by \(i\).

- If \(i\) is an identifier of a pointer object \(<i, n, i_1>\), and the object store contains the object \(<i_1, n_1, ... >\), then \(\text{nested}(i) = \{ n_1(i_1) \}\). The function for a pointer object returns the binder of the object that it points to.

- For a binder \(n(x)\) holds: \(\text{nested}(n(x)) = \{ n(x) \}\). For a binder \(\text{nested}\) returns this binder. As will be shown, this semantics is consistent with the typical understanding of auxiliary names introduced in queries.

- For a structure \(\text{nested}\) returns the union of the results of the \(\text{nested}\) function applied for elements of the structure:
  \(\text{nested}(\text{struct}\{ x_1, x_2, x_3, ... \}) = \text{nested}(x_1) \cup \text{nested}(x_2) \cup \text{nested}(x_3) \cup...\)

- For other arguments the result of \(\text{nested}\) is the empty set.
Fig. 3.9. Illustration of the function `nested`.

Examples of the results returned by the function `nested` (c.f. Fig. 2.2):

\[
\text{nested}(\text{struct}\{i_9, 55, i_{16}, "Maria"\}) = \{ \text{name}(i_9), \text{sal}(i_{11}), \text{address}(i_{12}), \text{worksIn}(i_{16}) \} \\
\text{nested}(\text{struct}\{\text{n("Doe")}, \text{sal}(2500), d(i_{22}) \}) = \{ \text{n("Doe")}, \text{sal}(2500), d(i_{22}) \} \\
\text{nested}(\text{bag}\{i_1, i_5, i_9\}) = \emptyset
\]

3.9 General Architecture of Query Processing

There are several views on architecture of query processing. Fig. 3.10 presents basic dependencies between three fundamental data structures involved in SBA: an object store, an environment stack and a query results stack. An object store contains volatile (non-shared) objects and persistent (shared) objects. Both stacks contain references to objects. Non-algebraic operators act on the query result stack and the object store and affect the environment stack. Query evaluation takes the state of the environment stack and the state of the objects store and puts query results on the query result stack.
In Fig.3.11 we present a more detailed view on the architecture, which involves more data structures (figures with dashed lines) and program modules (grey boxes). The most significant property of this architecture is that query analysis, optimization, compilation and processing are performed on the client side rather than on a server (in contrast to SBL-based relational DBMS). The major motivation is scalability: clients are usually much less overloaded than servers. Another motivation is that our queries are programming expressions; hence a lot of queries do not require access to the server or mix accesses to local data with accesses to servers. This architecture is implemented in our newest project ODRA. The architecture takes into account the subdivision of the storage and processing between client and server, strong typing and query optimization (by rewriting and by indices).
Below we present a short description of the architecture elements presented in Fig. 3.11. On the side of the client application we have the following elements.

- A source code of a query is created within the **software development environment**, which includes an editor, a debugger, storage of source programs, storage of compiled programs, etc.

- A **parser of queries** and programs takes a query source as input, makes syntactic analysis and returns a query/program syntactic tree.

- A **query/program syntactic tree** is a data structure which keeps the abstract query syntax in a well-structured form, allowing for easy manipulation (e.g. inserting new nodes or subtrees, moving some subtree to another part of the tree, removing some subtrees, etc.). Each node of the tree contains a free space for writing various query optimization information.

- The **strong type checker** takes a query/program syntactic tree and checks if it conforms to the declared types. Types are recorded within a client local metabase and within the

---

Fig. 3.11. Architecture of query processing with strong type checking and query optimization
metabase of persistent objects that is kept on the server. The metabases contain information from declarations of volatile object types (that are a part of source programs) and from a database schema. The module that organizes the metabases is not shown. The strong type checker uses two stacks, static ENVS and static QRES, which simulate actual execution of a query during compile time. Static stacks contain signatures of environments and signatures of results (will be explained much later). The static type checker has several other functions. In particular, it changes the query syntactic tree by introducing new nodes that allow, in particular, for automatic dereferences, automatic coercions, for resolving ellipses and for dynamic type checks (if static checks are impossible). The checker introduces also additional information to the nodes of the query syntactic tree that is necessary further for query optimization. The most important information is the level of the ENVS stack on which a particular name will be bound.

- **Static ENVS** (S_ENVS) - static environment stack (will be explained much later).
- **Static QRES** (S_QRES) - static result stack (will be explained much later).
- **Local metabase** - a data structure containing information of types introduced in source programs.
- **Optimization by rewriting** - this is a program module that changes the syntactic tree that is already decorated by the strong type checker. There are several rewriting methods that are developed and implemented for SBQL, in particular:
  - Performing calculations on literals.
  - Changing the order of execution of algebraic operators.
  - Application of the query modification technique, which changes invocations of views into view bodies. To this end, the optimization module refers to the register of views that is kept on the server.
  - Removing dead subqueries, i.e. subqueries that do not influence the final query result.
  - Factoring out independent subqueries, that is, subqueries whose result is not changed within some loop.
  - Factoring out weakly dependent subqueries, i.e. subqueries that depend on its superquery on an enumeration type.
  - Methods for queries that process large and small collections.
  - Shifting conditions as close as possible to the proper structure construction operator, e.g. shifting selection condition before a join or before a structure constructor.
  - Methods based on distributivity property of some query operators.
  - Removing unnecessary auxiliary names.
  - Perhaps other rewriting methods that are currently under investigation.
- **Optimization by indices** - this is a program module that changes the syntactic tree that is already decorated by the strong type checker. Changes concerns some subtrees that can be substituted by invocation of indices. To this end, the optimization module refers to the register of indices that is kept on the server. Changes depend on the kind of an index. The module can also be extended to deal with cached queries.
• **Interpreter of queries/programs.** It processes the optimized query syntactic tree and performs execution of the query. To this end it uses two run-time stacks, ENVS and QRES, which refer to **volatile (non-shared) objects** that are kept on the client and to **persistent (shared) objects** that are kept on the server. Object on the server are available through **object manager**, i.e. some API that performs everything on persistent objects that is needed.

On the side of the database server we have the following architectural elements:

• **Persistent (shared) objects** - this is a part of the object store commonly known as a database.

• **Object manager** - this is a low-level API that performs everything on persistent objects that is needed. Note that unlike SQL this API does not involve queries, but more atomic operations like “get first object *Emp***”, “get next object *Emp***”, etc.

• **Metabase of persistent objects** - this is a compiled database schema plus some additional information, e.g. necessary for optimization.

• **Processing persistent abstractions** (views, stored procedures, triggers) - essentially, this module contains all basic elements of the client side and extends them by additional functionalities.

• **Register of indices** and **register of views** are data structures that contain and externalize the information of created indices and created views. The information is used by the client for query optimization. Internally, this information is fulfilled by the **administration module**.

• **Administration module** - makes all operations that are necessary on the side of the server, e.g. introducing a new index, removing an index, introducing a new view, changing the database schema, etc.

In this architecture we assume that the query/program interpreter (which will be further formalized as the procedure *eval*) acts directly on a syntactic tree. Syntactic trees are the most convenient way to represent and to process **query evaluation plans**, perhaps, involving some optimizations. Cost-based query optimizers can generate several such query evaluation plans to take out one of them that is the most promising in terms of the anticipated query evaluation cost. The interpreter can act directly on a query syntactic tree (transformed by strong typing and optimization modules). However, we also assumed that this tree is further compiled to a bytecode and the interpreter acts on it. The advantage of this solution is that interpreter has no recursive procedures, hence is a bit more optimal. This solution was also implemented in Loqis. It is also possible that the tree will be further converted to the machine code. This solution was implemented in Linda, however, actually we see no essential advantages of it and a lot of disadvantages. We considered using in ODRA the Java bytecode, but because of its complexity and inadequacy to query languages we have eventually decided to use our own bytecode (called Juliet).

This view on the query processing architecture, although quite detailed, still can be augmented by new architectural elements, e.g. by a cost-based query optimizer, by introducing user sub-schemas, and many others. They will be refined in further parts of the SBQL description.
4 SBQL Syntax and Semantics for the AS0 Model

The Stack-Based Query Language (SBQL) is a formalized object-oriented query language in the SQL or OQL style. It contains semantic counterparts of all the constructs of these languages, but in essentially different configuration and semantic mechanisms. SBQL may concern many object store models, in particular AS0-AS3 models. Thus we claim that SBQL is the most complete and universal query language among all that can be found in the literature and practice.

The concrete syntax of SBQL is not fixed. In SBQL we strive to reduce syntactic considerations only to some abstract syntax. In concrete implementation any designer can invent an own concrete syntax that he/she considers the most adequate for the given purpose. So far the syntactic standard of SBQL does not exist. SBQL we consider as a theoretical frame or pattern for object-oriented languages, similar to the relational algebra or relational calculus. With no doubt, however, SBQL is much more powerful, strong, consequent and consistent in comparison to any such theory, including relational and object algebras.

The definition of SBQL explicitly involves the concept of state that is absent in major database theories. Lack of the state concept is a severe flaw of these theories, because it reduces the possibility to define many useful operators and from the very beginning forces some false view on query languages’ semantics. As we argued previously, the concept of the state and the naming-scoping-binding issues must lead to the concept of environment stack, which is absent in major database theories too.

Semantics of SBQL is expressed operationally by a machine, which accomplishes abstract implementation of SBQL constructs. Essentially, this decision is motivated by didactics, because operational semantics for so complex artifacts as query languages is much easier to grasp by the average reader. In our early work we tried to use denotational semantics, which is apparently “more elegant” (according to some theoreticians). We abandoned this specification style because it was illegible, non-intuitive and caused additional barriers in the understanding of implementation.

In SBQL we introduce a new kind of query operators, which we call “non-algebraic” for the tradition in mathematics which treats some operators (such as quantifiers) as non-algebraic ones. Selection, projection/navigation, join, quantifiers, ordering and transitive closures are all the non-algebraic operators that we introduce. Some of them (selection, projection, join, etc.) were introduced in the relational algebra as “algebraic” operators. However, their algebraization was at the cost of a bit unfair formal trick, where a part of their semantics (for instance, a selection predicate) was shifted to the informal meta-language of the mathematics. We have definitely rejected such a trick as frustrating, unacceptable and completely unnecessary. Mathematical solutions concerning semantics of query languages are not limited to such limited, simplistic and primitive theories as the relational algebra or relational calculus. In particular, the denotational semantics is much more adequate notion. However, as we have mentioned before, we do not follow strictly the mathematical method because essentially it has no advantages and has severe disadvantages. Our formal approach relies in very precise definition of an abstract machine that will perform operations implied by query operators. This level of formality is adequate to reason about any properties of query languages, including implementation of the language, possible methods of query optimization, strong typing, querying distributed databases and other issues.
4.1 SBQL Syntax

We assume that some elements of the set of atomic values $V$ introduced in the AS0 model have the external form, which allows representing them as sequences of characters in a query source code. The representation is sufficient to distinguish such an element by some lexer and then, to assign to it a proper value of $V$ (and an atomic type). Note that many elements of $V$, such as graphics, have no such source code representation. External representation elements we traditionally call literals\(^6\). The set of literals we denote $L$. We assume that the lexer accomplishes the function:

$$\text{lexval}: L \rightarrow V$$

Usually there are few types of literals: integer numbers, real numbers, strings, boolean values, and dates.

We assume that identifiers from the set $I$ introduced in the AS0 model have no such external representations, hence they cannot be directly used in source queries. The only way to deal with such identifiers is to use in queries external names from the set $N$. They are to be bound according to the rules explained in the previous chapter. In this way we are ready to define the simplest queries of SBQL:

- Any literal belonging to $L$ is a query. For instance, 2, 3.14, “Doe”, true are queries.
- Any name belonging to $N$ is a query. For instance, Emp, sal, worksIn, e, d are queries.

One may be surprised that a query such as sal is the query of its own rights. We understand that this can be unusual for the SQL users. However note that each query is evaluated relatively to the state, in particular, to the state of ENVS. If ENVS contains a binder sal( $x$ ), then the query sal makes a sense and its result can be formally determined as $x$.

Queries can be joined in more complex queries by operators. SBQL is perhaps the only practical query language which assumes that all operators are unary or binary. In this way the SBQL grammar is very simple. Moreover, the grammar in the maximal extent supports orthogonality and compositionality principles: queries can be freely connected by operators (providing typing constraints are not violated).

We subdivide binary operators into algebraic and non-algebraic. In this grammar we neglect parentheses assuming that they can be freely inserted if there are doubts concerning their order or precedence during the parsing. Then, the syntactic rules of SBQL are as follows:

- If $\Delta$ is a symbol denoting a unary algebraic operator and $q$ is a query, then $\Delta q$ is a query. Examples of unary algebraic operators are: count, sum, avg, max, median, -, log, sqrt, not, etc. Because SBQL is an abstract language, we do not fix which unary operators belong to SBQL and which do not belong. We simply assume that SBQL accepts any unary operator if some designer wants to introduce it.

- If $\Delta$ is a symbol denoting a binary algebraic operator and $q_1$, $q_2$ are queries, then $q_1 \Delta q_2$ is a query. Examples of binary algebraic operators are: =, <, >, +, -, *, /, and, or, intersection, concatenation, etc. As previously, we do not fix which binary algebraic operators belong to SBQL. Any of them will be accepted by us if it is necessary in

---

\(^6\)Note that the term literal is used in the ODMG standard in a different meaning, out of the traditional terminology of programming languages. We do not follow the ODMG terminology.
examples or in a particular implementation. For binary operators we assume the traditional infix notation (an operator will be written between its arguments).

- If $\theta$ is a symbol denoting a non-algebraic operator and $q_1, q_2$ are queries, then $q_1 \theta q_2$ is a query. Non-algebraic operators are the following: where (selection), dot (projection, navigation), join (dependent or navigational join), $\forall, \exists$ (universal and existential quantifiers), order by (sorting) and operators of transitive closures (will be introduced later). For quantifiers we also assume the traditional prefix notation: $\forall q_1 q_2$ and $\exists q_1 q_2$, which is equivalent to the universal infix notation $q_1 \forall q_2$ and $q_1 \exists q_2$.

- If $q$ is a query and $n \in \mathbb{N}$ then $q$ as $n$ is a query. The operator as is a unary algebraic operator parameterized by a name. This operator will be used in most cases which require defining an auxiliary name within a query. The operator as in other languages allows to define “correlation variables” (SQL, OQL), variables bound by quantifiers, cursors in for statements, etc. In ODMG OQL there are eight situations when one can define an auxiliary name. Our definition of the operator as will cover all such cases and more. As will be shown, the operator is orthogonal to any other operators and its semantics is very simple.

- If $q$ is a query and $n \in \mathbb{N}$ then $q$ group as $n$ is a query. This is another naming operator similar to as, but with different semantics and pragmatics.

- If $q_1, q_2$ are queries, then if $q_1$ then $q_2$ is a query. If $q_1, q_2, q_3$ are queries, then if $q_1$ then $q_2$ else $q_3$ is a query. The last query has three arguments, but it is a shorthand - it can be substituted by the composition of queries having one-argument and two-argument operators: if $q_1$ then $q_2$ else $q_3$ is equivalent to bag((if $q_1$ then $q_2$), (if not $q_1$ then $q_3$)).

- If $q_1, q_2, q_3, ...$ are queries, then struct($q_1, q_2, q_3, ...$) is a query. The operator struct has many arguments, but as in the previous case, it can be understood as a superposition of several binary operators; for instance, struct($q_1, q_2, q_3, q_4$) = struct(struct(struct(struct($q_1, q_2, q_3$), $q_4$)). In the SQL and OQL terminology the operator is called cartesian product; we do not use this term. The keyword struct can be skipped by default; ( $q_1, q_2, q_3, ...$) is equivalent to struct( $q_1, q_2, q_3, ...$).

- If $q_1, q_2, q_3, ...$ are queries, then bag($q_1, q_2, q_3, ...$) and sequence($q_1, q_2, q_3, ...$) are queries. As previously, the operators have many arguments, but can be understood as a superposition of several binary operators; for instance, bag($q_1, q_2, q_3, q_4$) = bag(bag(bag($q_1, q_2, q_3$), $q_4$).

These syntactic rules will be further extended by several more advanced constructs related to higher-level store models (AS1-AS3) and introducing programming constructs and abstractions such as procedures. Fig.4.1 contains the summary of the SBQL syntax for the AS0 model.
query ::= literal The set L
query ::= name The set N
query ::= unaryAlgOperator query Unary algebraic operators
unaryAlgOperator ::= count | sum | max | - | sqrt | not | ...
query ::= query binaryAlgOperator query Binary algebraic operators
binaryAlgOperator ::= =|<| >| +| -| *| /| and| or| intersect|...
query ::= query NonAlgOperator query Non-algebraic operator
NonAlgOperator ::= where | . | join | ∀ | ∃
query ::= ∀query query | ∃query query Alternative (traditional) syntax for quantifiers
query ::= query as name Name definition
query ::= query group as name Grouping and name definition
query ::= if query then query Conditional query
query ::= if query then query else query Another conditional query
querySeq ::= query | query, querySeq Sequence of queries
query ::= struct( querySeq ) | (querySeq) Structure constructor
query ::= bag( ) | bag( querySeq ) Bag constructor
query ::= sequence( ) | sequence( querySeq ) Sequence constructor

Fig.4.1. SBQL basic syntax

Examples of SBQL queries (c.f. Fig.2.2):

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 2000</td>
<td>Literal</td>
</tr>
<tr>
<td>2. Emp</td>
<td>Name</td>
</tr>
<tr>
<td>3. sal</td>
<td>Name</td>
</tr>
<tr>
<td>4. 2+2</td>
<td>Algebraic operator</td>
</tr>
<tr>
<td>5. sal &gt; 2000</td>
<td>Algebraic operator</td>
</tr>
<tr>
<td>6. Emp where (sal &gt; 2000)</td>
<td>Non-algebraic and algebraic operator</td>
</tr>
<tr>
<td>7. Emp where (sal &gt; 2000) . (name, (worksIn.Dept))</td>
<td>As before, plus projection and structure composition</td>
</tr>
<tr>
<td>8. ((Emp as e) where ((e.sal) &gt; (2000 + x + y))).e.name</td>
<td>Auxiliary name e</td>
</tr>
<tr>
<td>9. ((Emp as e) join ((e.worksIn.Dept) as d)) . (e.name, d.dname)</td>
<td>Dependent join with auxiliary naming, path expression, projection and struct</td>
</tr>
</tbody>
</table>
Looking at these examples we can see that SBQL is a very powerful query language. In fact, although its syntax is simple, it is much more powerful than OQL, LINQ and XQuery. All the features and examples of SBQL will be presented after explaining its semantics and pragmatics.

Note that in comparison to SQL and OQL we do not introduce operators for processing null values and a group by (having) clauses. In SBQL we avoid null values at all. Following Date, we consider null values as dangerous feature, creating more problems than it solves. Null values are substituted by absent (optional) data. Concerning the group by operator, an attempt to define consistent and universal semantics firmly convinced us that the operator is problematic (leading to semantic reefs and difficulties with query optimization) and unnecessary for object-oriented query languages - it can be substituted by other operators. We will show this further on examples.

4.2 Query Evaluation Procedure \textit{eval}

In the section devoted to the syntax, semantics and pragmatics of query languages we have described our approach to the operational semantics, in particular, to semantic rules implemented within an abstract evaluation machine. The machine has the form of the recursive procedure \textit{eval}. In this section we provide more detailed description of this procedure.

- The procedure \textit{eval} has a query as a parameter. The query is represented in the abstract syntax. In implementation the query has the form of a syntactic tree, which is the most convenient for quick and univocal recognizing its structure. The procedure \textit{eval} makes no changes to the syntactic tree.

- In real implementation, before the procedure \textit{eval} is called, the syntactic tree is processed by a strong type checker and by optimization modules, see Fig.4.1 (\textit{eval} is some abstraction over the interpreter of queries/programs module). The strong type checker removes ambiguities that may occur in the source code, for instance, maps literals into values of corresponding types, resolves automatic dereferences, coercions and ellipses, and makes other changes to the tree. Then, the optimization module can make significant changes of the tree aiming query optimization. All these changes, however, do not influence the result returned by the query, hence we neglect them during our description of \textit{eval}.

- The procedure \textit{eval} will operate on three basic data structures that are introduced in SBA: the object store, the ENVS stack and the QRES stack, see Fig.4.1.

- It may change the state of ENVS, but for queries with no side effects the initial state and the final state of ENVS is the same.
• Assuming no side effects in queries, the procedure makes no changes within the object store.

• The procedure makes no change in the initial part of the QRES state, i.e. sections that were on QRES when the procedure started are retained without changes. At the end of the procedure, QRES is augmented by a new top section with the result of the processed query, Fig.4.2.

• The result from the top of QRES is then “consumed” by some agent external to the query, for instance, by a print procedure, by a user graphical interface, by an update command, etc. After such operation the stack returns to the initial state.

Fig.4.2. QRES changed by the procedure eval and by a query result consumer

Fig.4.3 presents an example of a syntactic tree for a simple query. A node of the tree contains information on its kind and may contain the information from the query. Each node contains also some free space (a grey box) that can be filled in by a type checker or by optimization modules. We explain the kind and role of this information later. A tree may contain some elements that are implicit in the query, for instance, the algebraic dereference (deref) operator.

Fig.4.3. Syntactic tree of the query: Emp where name = “Poe”

The procedure eval follows the orthogonality and compositionality principles, which say (in the case of queries) that the semantics of a query is some function of the semantics of its direct subqueries. The principles imply simple and intuitive recursive definition of the semantics. The principles have also great meaning for generalization of query optimization methods.
The definition of `eval` considers a few cases that are recognized in the syntactic tree. We present the procedure in the form of a pseudo-code. Our goal is to illustrate the formal semantics rather than to present details of all the operators that we would like to introduce in SBQL. For the purposes of standardization the procedure `eval` can be refined precisely, but this requires some agreement on the pseudo-code that will be used for the specification. The agreement is similar to the standardization of syntax in the form of BNF grammars.

The simplest SBQL queries are literal and names. During parsing or type checking literals are changed into values of proper types. This operation can be written as application of the function `lexval: L → V`, where `L` is the set of literals, `V` is the set of object store values. Concerning names occurring in queries, they are bound on `ENVS` and the corresponding function is `bind: N → Result`, where `N` is the set of all names, `Result` is the set of all query results, as defined previously. Other cases include unary algebraic operators, binary algebraic operators, naming operators (unary algebraic operators parameterized by a name), non-algebraic operators and structure/collection construction operators. The corresponding parts of the `eval` procedure will be explained subsequently for particular operators or their kinds. Below we present the beginning and some frame of the procedure. Stack operators `push`, `pop`, `top` will be parameterized by the name of the stack. We also distinguish symbols of operators occurring in queries and the operators themselves as mathematical beings. Operators will be underlined: if `S` is a symbol denoting an operator, the \( S \) is the operator itself.

```plaintext
procedure eval( q : query )
begin
  temp1, temp2: Result;  //two local variables for temporary results of the type Result
  case q is recognized as literal \( l \in L \):
    push( QRES, lexval( l ) );
  case q is recognized as name \( n \in N \):
    push( QRES, bind( n ) );
  case q is recognized as \( \Delta q_1 \):
    // the query \( q \) consists of a unary algebraic operator \( \Delta \) applied to a subquery \( q_1 \)
    eval( q_1 );
    temp1 := top( QRES ); pop( QRES );
    push( QRES, \( \Delta\) ( temp1 ) );
  case q is recognized as \( q left \Delta q right \):
    // the query \( q \) consists of a binary algebraic operator \( \Delta \) joining subqueries \( q left \) and \( q right \)
    eval( q left );
    eval( q right );
    temp2 := top( QRES ); pop( QRES );
    temp1 := top( QRES ); pop( QRES );
    push( QRES, \( \Delta\) ( temp1, temp2 ) );
  case q is recognized as \( q_1 as n \): //the query \( q \) is the assignment of an auxiliary
    //name \( n \) to elements of the collection returned by \( q_1 \)
    ... // semantics will be explained later
  case q is recognized as \( q_1 group as n \): //the query \( q \) is the assignment of an auxiliary
    //name \( n \) to the entire result returned by \( q_1 \)
    ... // semantics will be explained later
  case q is recognized as \( q left 0 q right \): // the query \( q \) consists of a non-algebraic
    //operator \( 0 \) joining subqueries \( q left \) and \( q right \)
```
... // semantics will be explained later

**case** $q$ is recognized as **if** condition **then** $q_1$: //conditional query: if condition is false,
  //push the empty bag on QRES
  eval( condition );
  temp1 := top( QRES ); pop( QRES );
  if ( temp1 ) then eval( $q_1$ ) else push(QRES.bag() );

**case** $q$ is recognized as **if** condition **then** $q_1$ **else** $q_2$ //conditional query
  eval( condition );
  temp1 := top( QRES ); pop( QRES );
  if ( temp1 ) then eval( $q_1$ ) else eval( $q_2$ );

**case** $q$ is recognized as **struct**( querySeq ): //constructor of structures
  ... // semantics will be explained later

**case** $q$ is recognized as **bag**( querySeq ): //constructor of bags
  ... // semantics will be explained later

**case** $q$ is recognized as **sequence**( querySeq ): //constructor of sequences
  ... // semantics will be explained later

...// perhaps other cases
**end**

In the following we subdivide SBQL query operators into *algebraic* and *non-algebraic*. The subdivision is based on a very simple rule:

- **Algebraic operators** do not deal with the ENVS stack; they act on the QRES stack only. Algebraic operators can be unary, binary and perhaps may have more arguments. In SBQL all operators having more than two arguments can be reduced to the composition of unary and binary operators. For instance, the ternary operator **if** condition **then** $q_1$ **else** $q_2$ can be understood as a composition of three binary and one unary operators: **bag**( **if** condition **then** $q_1$, **if** not condition **then** $q_2$ ); providing $q_1$ and $q_2$ return bags. Similarly, **bag**( $q_1$, $q_2$, $q_3$, $q_4$ ) can be understood as **bag**( $q_1$, **bag**( $q_2$, **bag**( $q_3$, $q_4$ ) ) ). If an algebraic operator $\Delta$ is binary, then in the query $q_{left}$ $\Delta$ $q_{right}$ the order of evaluating $q_{left}$ and $q_{right}$ does not matter. We would like to emphasize (due to frequent misunderstanding) that we are talking on the order of evaluation, not about the order of arguments. In general, $q_{left}$ $\Delta$ $q_{right}$ is not equivalent to $q_{right}$ $\Delta$ $q_{left}$ (this holds of course for some operators, but not for all), but this means that we can call the eval procedure first to $q_{right}$ and then to $q_{left}$ and still the semantics of the query can be correctly determined.

- **Non-algebraic operators** deal with both ENVS and QRES. Non-algebraic operators are the core of SBA and SBQL. The fundamental difference with algebraic operators is that in the query $q_{left}$ $\theta$ $q_{right}$, where $\theta$ is a non-algebraic operator, the order of evaluating queries $q_{left}$, $q_{right}$ is essential. The procedure eval must be first called on the $q_{left}$ argument, and then, the procedure eval is called many times for the $q_{right}$ argument. We present this part of the eval procedure a bit later, after discussion and introducing some new notions.

The definition of other algebraic operators that are present in the above procedure eval body is simple and straightforward; we return to them later.
4.3 Algebraic Operators

In the description of the query evaluation procedure \textit{eval} we have presented the operational semantics of typical algebraic operators that we would like to introduce in SBQL. We do not fix which operators belong to SBQL and which do not belong. We assume that SBQL contains all operators which can be necessary to illustrate some topics by examples and all operators that developers of some SBQL interpreter wish to introduce. There are myriads of algebraic operators, thus fixing some particular subset of them is not reasonable. All of them can be defined and implemented on the same semantic and implementation basis. Moreover, after implementing a sufficiently rich set of operators, next ones can be implemented in quarters or hours. Determining the set of operators that belong to SBQL is of course important if some organization wish to have SBQL as a standard. So far such a goal looks as very distant (if real at all).

There is a subtle difference between operators that are \textit{built in} the language and operators that are \textit{added on}. Build in operators are the property of a particular interpreter. Added on operators are possible if an interpreter accepts external libraries (of procedures, functions or classes) in some general way, e.g. as dynamically linked functions a la MS Windows dll-s. Because eventually we extend SBQL to fully-fledged programming language, it is also possible to have some added on functionalities written in SBQL (in particular, procedures, functions, classes/methods and views). The operators can perform complex computations, for instance, statistical or data mining functions. In general, we are not interested in how complex is the implementation of a particular \textit{build in} or \textit{added on} operator.

To simplify the explanation, besides the description of algebraic operators in the form of the \textit{eval} procedure we will use a more abstract (actually, denotational) view on semantics by introducing the function \textit{result}. We consciously take the same function name as the name of the domain \textit{Result} because of the similarity of the concepts. The function \textit{result} abstracts from the QRES stack. It returns the final result of the query being its argument. In particular, we can take the following semantic rules:

- If $\Delta$ is a symbol denoting a unary algebraic operator, then
  \[ \text{result}(\Delta \, \text{q}_1) = \Delta(\text{result}(\text{q}_1)) \]
  As before, $\Delta$ is the operator denoted by the symbol $\Delta$.

- If $\Delta$ is a symbol denoting a binary algebraic operator, then
  \[ \text{result}(\text{q}_1 \, \Delta \, \text{q}_2) = \Delta(\text{result}(\text{q}_1),\text{result}(\text{q}_2)) \]

- And so on.

Note that the function \textit{result} is implicitly parameterized by the state, i.e. by the object store and ENVS. At this level of abstraction we need not to talk about the stack QRES at all. However, this is only some didactic facility; formal semantics is to be given by the procedure \textit{eval}.

In general, we assume that all algebraic operators have no side effects, i.e. they are pure functions. Side effects may concern, for instance, updating operations in a database, some visual effects on the user screen, some writings to files, etc. Side effects of operators occurring in query languages usually undermine query optimization and frequently are difficult to understand by the programmers. Nevertheless, operators with side effects could be necessary for many applications. However, their use must be disciplined. This will be possible in imperative programming extensions of SBQL.
4.3.1 Operators and Comparisons for Primitive Types

- We assume generic and overloaded comparison operator \(=\) (and its complement \(\neq\)) for all the primitive types that we introduce. The operators have the obvious meaning. Examples: \(2 = 2, x \neq y, \text{sal} \neq 2000, \text{name} = \text{“Doe”},\) etc. We assume that operators \(=\) and \(\neq\) can also act on complex values (e.g. structures, sets, collections), however with the caution that their meaning must be univocally defined. Operators may also act on object references, but the comparison is shallow, e.g. \(i_1 = i_2\) returns true if \(i_1\) and \(i_2\) refer to the same object. We do not assume any form of a deep comparison, i.e. comparison of objects that the argument references refer to. (Sometimes, however, deep comparisons may be enforced by the strong typing system and automatic dereferences, but this is another story that we explain elsewhere.) The operators are undefined for arguments of different types, for instance (real) \(3.14 = \text{(int)} 3\) is illegal. Such comparisons, however, will be possible due to automatic coercions, which will be introduced later.

- For primitive types having natural linear ordering we introduce generic and overloaded comparison operators \(<, \leq, >, \geq\). These operators we can apply to integer numbers, real numbers (float), strings (assuming some predefined lexicographical order, perhaps corresponding to the current character coding table), dates and times (assuming the linear time axis), etc. Operators are undefined (not allowed) for arguments of different atomic types and are undefined for complex values and identifiers. Examples: \(2 < 3, x \leq y, \text{sal} \geq 2000, \text{name} > \text{“Doe”},\) etc.

- For primitive numerical types we introduce typical arithmetic operators \(+, -, *, /\). Operator - is unary and binary. Their meaning is typical.

- We assume several string-oriented operators and comparisons, such as the concatenation of strings (the overloaded + operator), is_a_substring and is_a_superstring. We also provide parameterized string operators, such as taking a substring from i-th to j-th character (perhaps in the C/C++ or Java convention), some functions, such as changing strings to capital letters, counting the size of a string, etc.

- We assume several build in functions acting on numerical values, such as the integer part of a real number, rounding a real to integer, square root, logarithm, power, etc. The syntax and meaning e.g. as in Pascal.

- We also provide some functions for dates, for instance, change the date into the number of days since 1900.01.01 and v/v, changing data formats, changing the time to the number of milliseconds, etc.

The above operators and comparisons cause some temptation of the developers of query language interpreters to extend their usual meaning to arguments being collections. For instance, one wants to define \(\text{result( bag(3, 7) + bag(1, 4, 5)) = bag\{ 4, 7, 8, 8, 11, 12 \}}\). This philosophy resulted in SQL in special comparison operators, such as =\(\text{any}\) and =\(\text{all}\). Taking such extensions of popular operators with commonly understood semantics is however rather awkward and error prone. In many cases the programmer may have problems with understanding what they actually mean. They also weaken the strong typing system.

Thus we discourage such extensions of their semantics. Collections are to be processed by other operators, in particular, quantifiers. The commonly understood operators, such as \(=, \neq, <, \leq, >, \geq, +, -, *, /\), should not extend their meaning. Using of them should be disallowed for arguments being collections of primitive types. SBQL provides options that make it possible to achieve the desired results without such extensions of the semantics of these operators.
4.3.2 Aggregate Functions, Removing Duplicates

Aggregate functions act on arguments being collections (bags or sequences) of integer or real numbers. For empty collections the functions are undefined (not allowed). We assume the typical set of numeric aggregate functions known from SQL:

- **sum** - the sum of argument numbers; for instance, \( \text{result}( \text{sum}( \text{bag}(3, 6, 7)) ) = 16 \), \( \text{sum}(\text{Emp.sal}) \) calculates the total sum of employees’ salaries.

- **avg** - the arithmetic average of argument numbers, for instance, \( \text{avg}(\text{bag}(3, 6, 7)) \) returns \( 5.333... \), \( \text{avg}(\text{Emp.sal}) \) calculates the average of employees’ salaries.

- **max** - the maximal number from the argument collection.

- **min** - the minimal number from the argument collection.

Several next similar functions can be useful for some applications; for instance, median, geometric average, variance, etc.

This set is usually augmented by:

- **count** - the size on any collection; the function is defined for any kind of a collection, any type of its elements and for empty collections.

- **exists** - test for existence; it returns \textit{true} if the argument collection is not empty and \textit{false} if it is empty.

- **distinct** - the function removing duplicates from the argument collection, for instance, \( \text{result}(\text{distinct}(\text{bag}(3, 6, 3, 3, 6, 7))) = \text{bag}(3, 6, 7) \). For sequences \( \text{distinct} \) returns the sequence where each next repetition of an element is removed, for instance, \( \text{result}(\text{distinct}(\text{sequence}(7, 7, 3, 6, 3, 3, 6))) = \text{sequence}(7, 3, 6) \).

In SQL functions \textit{sum}, \textit{avg}, \textit{max}, \textit{min} and \textit{count} have two semantic faces. If they are not under the scope of the \textit{group by} clause, they have the usual semantics, as presented above. If they are under the scope, they calculate their values for groups determined by the \textit{group by} clause and in this semantic sense can be used within the \textit{select} and \textit{having} clauses. Moreover, SQL assumes quite unusual (apparently more friendly) syntax, where names of these functions are under the \textit{select} clause; for instance, to calculate the average salary of employees one has to write \( \text{select avg(sal)} \text{ from Emp} \) rather than more logical formulation \( \text{avg(\text{select sal from Emp})} \). (This syntax is also an option in ODMG OQL.)

In SBA and SBQL we have rejected such doubtful syntactic and semantic inventions. No function from the above list will have two different semantic faces. The regular syntax of their use will not be violated. Moreover, we do not introduce the \textit{group by} operator at all, for its severe disadvantages, such as non-orthogonality, semantic reefs that it implies and implementation-oriented flavor that makes difficulty for query optimization (we discuss this later). All the pragmatic goals that can be achieved through the \textit{group by} clauses can be achieved in SBQL through other operators, in easier, more logical and more consistent way. We will show that on examples. Lack of the \textit{group by} clause causes that specific problems with a special conditional clause (\textit{having}), non-orthogonality, semantic reefs and query optimization no more exist.

In SQL there is also semantic schizophrenia concerning null values. Despite big words on the meaning of null values and special constructs for serving them in queries, aggregate functions \textit{sum}, \textit{avg}, \textit{max}, \textit{min} ignore them totally. It is not quite clear how they are treated by \textit{count}, \textit{exists} and \textit{distinct}. The \textit{group by} clause collects all null values in a single group, what makes
4.3.3 Operators and Comparisons on Collections

These operators come from the set theory, which introduce such operators as union of sets, intersection of sets, difference of sets, cartesian product, set containment ($\subset$, $\subseteq$), membership of an element in a set (operator $\in$), and so on. Actually, we have no sets as data structures, but bags and sequences; hence we have to adopt these set-theoretic operators to a bit different situation. We have already introduced some of these operators, in particular:

- **struct** $(A, B)$ - the constructor of structures, which in the SQL jargon is known as *cartesian product*. If query $A$ returns $\text{bag}\{a_1, a_2, \ldots\}$ and query $B$ returns $\text{bag}\{b_1, b_2, \ldots\}$, then the query $\text{struct}(A, B)$ returns $\text{bag}\{\text{struct}(a_1, b_1), \text{struct}(a_1, b_2), \ldots, \text{struct}(a_2, b_1), \text{struct}(a_2, b_2), \ldots\}$. If $A$ returns an individual element $a$, and $B$ returns an individual element $b$, then $\text{struct}(A, B)$ returns an individual element $\text{struct}(a, b)$. In all other cases individual elements are converted to one-element bags. Our syntax allows us to drop *struct* and to write simply $(A, B)$. We assume that $\text{struct}\{a, \text{struct}(b, c)\}$ is equivalent to $\text{struct}\{\text{struct}(a, b), c\}$ and is equivalent to $\text{struct}\{a, b, c\}$. We also assume that a structure with one element is equivalent to this element: $\text{struct}\{a\} = a$. Similar properties are then induced on the level of the query language semantics: in particular, query $\text{struct}(\text{struct}(A, B), C)$ is equivalent to query $\text{struct}(A, \text{struct}(B, C))$ and is equivalent to query $\text{struct}(A, B, C)$. If $A$ returns an individual element, then $\text{struct}(A)$ is equivalent to $A$. As follows from the above, $\text{struct}(A, B, C, D, E, \ldots)$ is a shorthand for composition of several binary struct operators: $\text{struct}(A, \text{struct}(B, \text{struct}(C, D, \text{struct}(E, \ldots))))$. If $A$ or $B$ returns an empty bag, then $\text{struct}(A, B)$ returns an empty bag. It is quite unclear what $\text{struct}(A, B)$ has to return if $A$ and/or $B$ return sequences; hence we propose to leave such cases undefined (i.e., to forbid them). Unlike SQL, we do not limit the cases where our “cartesian product” can occur - it may occur in any place of a query if it makes sense for the programmer and the strongly typed is not violated.

- **bag** $(A, B)$ - the constructor of bags, which in the SQL jargon is known as *union of tables (relations)*. If query $A$ returns $\text{bag}\{a_1, a_2, \ldots\}$ and query $B$ returns $\text{bag}\{b_1, b_2, \ldots\}$, then the query $\text{bag}(A, B)$ returns $\text{bag}\{a_1, a_2, \ldots, b_1, b_2, \ldots\}$. If $A$ or $B$ returns an individual element, it is treated as a one-element bag. We assume that $\text{bag}\{a, \text{bag}\{b, c\}\}$ is equivalent to $\text{bag}\{\text{bag}\{a, b\}, c\}$ and is equivalent to $\text{bag}\{a, b, c\}$. We also assume that a bag with one element is equivalent to this element: $\text{bag}\{a\} = a$; our strong typing system will be prepared to handle such cases correctly. Similar properties are then induced on the level of the query language semantics: in particular, query $\text{bag}(\text{bag}(A, B), C)$ is equivalent to queries $\text{bag}(A, \text{bag}(B, C))$ and $\text{bag}(A, B, C)$. If $A$ returns an individual element, then $\text{bag}(A)$ is equivalent to $A$. As before, $\text{bag}(A, B, C, D, E, \ldots)$ is a shorthand for composition of several binary bag operators: $\text{bag}(A, \text{bag}(B, \text{bag}(C, \text{bag}(D, \text{bag}(E, \ldots))))$. If $A$ or $B$ returns an empty bag, then it does not influence the result bag. It is quite unclear what $\text{bag}(A, B)$ has to return if $A$ and/or $B$ return sequences; hence (as for struct) we propose to leave such cases undefined (to forbid them). As for struct, the operator may occur in any place of a query if it makes sense for the programmer and the strongly typed is not violated.
• **sequence**\((A, B)\) - the constructor of sequences, known as concatenation of sequences. If query \(A\) returns \(\text{sequence}\{a_1, a_2, \ldots\}\) and query \(B\) returns \(\text{sequence}\{b_1, b_2, \ldots\}\), then \(\text{result}(\text{sequence}(A, B)) = \text{sequence}\{a_1, a_2, \ldots, b_1, b_2, \ldots\}\). If \(A\) or \(B\) returns an individual element, it is treated as a one-element sequence. We assume that \(\text{sequence}\{a, b, c\}\) is equivalent to \(\text{sequence}\{\text{sequence}\{a, b\}, c\}\) and to \(\text{sequence}\{a, b, c\}\). We also assume that a sequence with one element is equivalent to this element: \(\text{sequence}\{a\} = a\); as for bags, our strong typing system will be prepared to such cases. Similar properties are then induced on the level of the query language semantics: in particular, query \(\text{sequence}(\text{sequence}(A, B), C)\) is equivalent to queries \(\text{sequence}(A, \text{sequence}(B, C))\) and \(\text{sequence}(A, B, C)\). If \(A\) returns an individual element, then \(\text{sequence}(A)\) is equivalent to \(A\). As before, \(\text{sequence}(A, B, C, D, E,\ldots)\) is a shorthand for composition of several binary sequence operators: \(\text{sequence}(A, \text{sequence}(B, \text{sequence}(C, \text{sequence}(D, \text{sequence}(E,\ldots))))))\). If \(A\) or \(B\) returns an empty sequence, then it does not influence the result sequence. It is quite unclear what \(\text{sequence}(A, B)\) has to return if \(A\) and/or \(B\) return bags; hence (as for struct and bag) we propose to leave such cases undefined (to forbid them). As for struct and bag, the operator may occur in any place of a query if it makes sense for the programmer and the strong typing is not violated.

Examples of application of \(\text{struct}\), \(\text{bag}\) and \(\text{sequence}\) are the following:

\[
(\text{Emp}, \text{Dept}) \quad \text{where name = dname} \\
\text{struct}(\text{Emp}, \text{Dept}) \quad \text{where name = dname}
\]

\[
(\text{Emp where name = “Poe”}). (\text{sal, worksIn}) \\
\text{struct}(\text{Emp where name = “Poe”}). \text{struct} (\text{sal, worksIn})
\]

\[
\text{struct} (\text{name(“Kim”), sal(3000), worksIn(\text{Dept where dname = “Trade”})})
\]

\[
\text{bag}((\text{Emp where sal < 1000}), ((\text{Dept where dname = “Trade”}).\text{employs}.\text{Emp}))
\]

\[
\text{sequence (“one”, “two”, “three”, “four”})
\]

Besides these operators, we consider to introduce next ones:

• **in** - bag or sequence inclusion operator. A query \(A\ \text{in} B\) returns \(\text{true}\) if each element of the bag that is returned by \(A\) can be found in the bag returned by \(B\). For sequences, the query \(A\ \text{in} B\) returns \(\text{true}\) if the sequence returned by \(A\) is a sub-sequence of the sequence returned by \(B\). It is quite unclear what \(A\ \text{in} B\) has to return if \(A\) returns a bag and \(B\) returns a sequence (or v/v); hence we propose to leave such cases undefined (i.e., to forbid them). If \(A\) returns an empty bag or an empty sequence, the query \(A\ \text{in} B\) returns \(\text{true}\). If \(A\) returns an individual element, it is treated as a one element bag or one element sequence (depending on what returns \(B\)).

• **contains** - a reverse bag or sequence inclusion operator; \(A\ \text{contains} B\) is equivalent to \(B\ \text{in} A\).

• **intersect** - makes a sense for bags only. A query \(A\ \text{intersect} B\) returns a bag consisting of elements that occur both in the bag returned by \(A\) and in the bag returned by \(B\). Because a bag can contain the same element \(x\) many times, the result of \(A\ \text{intersect} B\) contains the minimal number of \(x\) from the numbers of occurrences of \(x\) in \(A\) and in \(B\). If \(A\) or \(B\) returns an individual element, it is treated as one-element bag.

• **subtract** - difference of bags. A query \(A\ \text{subtract} B\) returns a bag of elements that occur in the bag returned \(A\) and do not occur in the bag returned by \(B\). If element \(x\) occurs in the result of \(A\ k_1\) times, and in the result of \(B\ k_2\) times, then the result \(A\ \text{subtract} B\) contains \(x\) if \(k_1 - k_2 > 0\) and the number of occurrences of \(x\) in the result is \(k_1 - k_2\).
• i-th element of a sequence. We denote this operator \( \text{query1}[\text{query2}] \), where \( \text{query1} \) returns a sequence, and \( \text{query2} \) returns an integer number. Unlike C, C++, Java and ODMG, we propose to return to the oldest tradition where the first element of a sequence is indexed by the number 1 (not by 0). For instance, \( \text{result( sequence(7, 6, 3)[2] )} = 6 \). Note that the operator makes no sense for bags.

• Elements of a sequence from i-th to j-th. Similarly as before, we denote this operator \( \text{query1}[\text{query2}.. \text{query3}] \), where \( \text{query1} \) returns a sequence, \( \text{query2} \) returns a lower index and \( \text{query3} \) returns an upper index of the sequence elements that are taken as the result. For instance, \( \text{result( sequence(7, 6, 3,11, 5, 6, 7, 34, 3)[4..7] )} = \text{sequence}\{11, 5, 6, 7\} \)

There are some less important operators, such as proper sub-bags and sub-sequences, continuous sub-sequences, cutting heads and tails of sequences, i-th element from the end of a sequence, taking a random single element from a bag, etc. We can introduce them if there will be some significant applications.

### 4.3.4 Coercions and Dereferences

Coercions are functions that change the types and representation of values. In a lot of cases coercions are implicit, to avoid annoying style of programming. For instance, if \( x \) is an integer number and \( y \) is a real number, then in the query \( x + y \) the value returned by \( x \) is automatically coerced to a real number. Similarly, in the query \( \text{Emp.(name} + \text{“ earns “ } + \text{sal}) \) the operator + is recognized as concatenation of strings, hence integer values returned by \( \text{sal} \) are implicitly coerced to strings. The presence of implicit coercions is sometimes called *ad hoc polymorphism*.

Another kind of implicit coercions concerns changing bags or sequences into single elements, and v/v. For instance, in SQL a select clause can occur within a where clause, but in this case the result of the select clause is automatically coerced to an individual value. Such coercions are typical for SBQL, which unifies a structure, bag or sequence having one element \( x \) and this element \( x \). For instance, one can use a query (get employees earning more than Kim):

\[
\text{Emp where sal > } (\text{Emp where name} = \text{“Kim”}).\text{sal}
\]

The query assumes implicit coercion of the bag returned by \( (\text{Emp where name} = \text{“Kim”}).\text{sal} \) into a single value (of the Kim’s salary). If this is impossible, because the company does not employ a person named Kim, or the company employs more than one person named Kim, the dynamic strong type checker will return a typing error (exception).

A consistent and universal handling of implicit coercions requires a static strong typing system. During type checks of a query it recognizes the necessity of a coercion, inserts a proper explicit coercion operator into the query syntactic tree and instantly augments the tree with some parts performing dynamic type checks. In this way the static (compile-time) strong typing system delegates corresponding type checks to run-time. For instance (see Fig.2.1), considering the query \( \text{Dept where location} = \text{“Paris”} \) (where location returns a bag), the syntactic tree of this query is augmented by a part performing the check if indeed location returns a single value; this part returns a type error if the bag contains two or more elements. We explain this issue in detail when we will come to the specification of the SBQL typing system.

Another commonly assumed kind of implicit coercions are *implicit dereferences*. Assuming an object \(<i, n, v>\), the dereference of \( i \) returns \( v \). For instance (see Fig.2.1), for the query \( \text{Emp where name} = \text{“Poe”} \) the subquery name returns a reference; it is automatically dereferenced to a value of the object pointed to by this reference. This is the job of the static...
A strong type checker, which must properly change the syntactic tree of a query, see Fig.4.3. Usually, the implicit dereference concerns values of primitive types. However, dereferences of identifiers of complex objects can be also consistently defined; we explain this issue later.

For the programmers convenience coercions and dereferences can be explicitly introduced as algebraic operators. The typical syntactic convention for that is known from languages such as C, C++, Java, etc. as cast. We use this convention for run-time explicit conversions of types. Casts will be written in the syntax

\[
query ::= \text{(typeName)}\ query
\]

Note however that in contrast to C, where arbitrary casts are allowed (and this is rather a compile-time facility), we assume that our cast are run-time conversions, thus can be allowed or disallowed. Sometimes they may cause compile time-typing errors. For instance \((\text{int})\text{Emp}\) is disallowed and will be discovered as a type error. In some cases coercions may cause a run-time error. For instance, the query \((\text{int})X\), where \(X\) is of type \text{string}, causes conversion of the string stored at \(X\) into integer, but this is of course not always possible.

We also use casts with collection constructors. For instance, \((\text{sequence})\text{Emp}\) changes the bag returned by \text{Emp} into a sequence with an arbitrary order of elements. Similarly, we can use \((\text{bag})\text{query}\) to convert a sequence into a bag; the operator has no effect if the \text{query} returns bag.

In the following (in the AS1, AS2 and AS3 models) such casts will also be used to convert an object into an object of a more specific or a more general class. For instance, \((\text{Student})\text{(Emp where sal > 2000)}\) converts references to objects of employees earning more than 2000 into references to \text{Student} objects, but this conversion rejects such \text{Emp} objects that are not at the same time \text{Student} objects.

In some cases there is also a need for explicit dereference operators. This will be possible by the operator deref that takes a single references or a collection of references and then returns a corresponding value or a collection of values. For instance (see Fig.2.1):

\[
\begin{align*}
\text{result}(\text{Emp.name}) & = \text{bag}\{i_2, i_6, i_{10}\} \\
\text{result}(\text{deref(Emp.name)}) & = \text{bag}\{\text{"Doe"}, \text{"Poe"}, \text{"Lee"}\}
\end{align*}
\]

### 4.3.5 Conditional Queries

The semantics of conditional queries is rather obvious:

\[
\begin{align*}
\text{result}(\text{if } q_1 \text{ then } q_2) & = \text{if result}(q_1) = \text{true} \text{ then result}(q_2) \text{ else bag}\{\} \\
\text{result}(\text{if } q_1 \text{ then } q_2 \text{ else } q_3) & = \text{if result}(q_1) = \text{true} \text{ then result}(q_2) \text{ else result}(q_3)
\end{align*}
\]

**Example**

\[
\text{if count(Emp) < 10 \ then \ Emp.name \ else \ (bag)((sequence)(Emp.name)[1..10])}
\]

As in Oracle, we can also consider a more general case of conditional queries that take the form of a switch operator.

### 4.3.6 Defining Auxiliary Names

Auxiliary names appear as a property of query languages in connection with the relational calculus, considered a mathematical basis for relational query languages. The relational calculus introduced such names as “variables” iterating over relations (tuple relational
calculus) or over columns (domain relational calculus). The variables can be then used in selection conditions, in a clause forming the output from a query, etc. Similar variables occur in predicate calculus and in lambda calculus. QUEL, one of the first relational query languages, makes the use of such “tuple variables” obligatory. Query-By-Example is considered a language based on the domain relational calculus.

SQL introduces auxiliary names, but their use is not obligatory. They may appear in a from clause after names of relations. Sometimes they are called “synonyms” of relation names, but this is improper association, because in where clauses and in select clauses they behave just as variables ranging over relations. In other terminology they are called “correlation variables” or “correlation names”. For instance, assume the relational schema (worksIn is a foreign key for d#, boss is a foreign key for e#):

Emp(e#, name, job, sal, worksIn)
Dept(d#, dname, location, boss)

We can formulate the request “get designers earning more than their bosses” as the following SQL query:

```sql
select x.name from Emp x, Dept y, Emp z
where x.job = 'designer' and x.worksIn = y.d#
and y.boss = z.e# and x.sal > z.sal
```

This query defines three correlation names x, y and z. The use of x and z cannot be avoided because both refer to the same table Emp. However, the use of such variables in SQL is not obligatory and one can formulate the same query without the variable y:

```sql
select x.name from Emp x, Dept, Emp z
where x.job = 'designer' and x.worksIn = Dept.d#
and Dept.boss = z.e# and x.sal > z.sal
```

Sometimes this is explained by the rule that each relation named R has by definition the correlation name named R.

This idea of auxiliary naming in SQL seems to be quite semantically clear. However, in other query languages the situation is not obvious. For instance, such variables are necessary in quantified predicates as “variables bound by quantifiers”, in for each iterators as a variable iterating over some calculated domain, in group by clauses, in order by clauses, etc. Many contexts when such “variables” may appear and complex data structures that are to be served by such variables cause the necessity to explain their semantics on the more general ground that is done in the relational calculus.

An example of difficulties with correct definition of the semantics of such auxiliary names is supplied by the ODMG standard. Consider the following OQL query, p.104 of [ODMG00]:

```sql
select * from Students as x, x.takes as y, y.taught_by as z
where z.rank = "full professor"
```

The query defines three names x, y, z, considered “iteration variables”. According to the scope rules (p. 112), their scope is limited to this query (they have no meaning outside it). But it appears (p. 105) that the type returned by this query is:

```
bag<struct (x:Students, y:Section, z:Professor)>
```

The semantic qualification of these names is changed: instead of being “iteration variables” they become structure field labels. Because the output of the query can be used in another (outer) query, the names can be used outside the query where they have been defined (which is inconsistent with the assumed scope rules).
This example raises very fundamental questions:

1. Which formal model makes it possible to change iteration variables into structure field labels? Clearly, the relational calculus does not provide such situations.

2. What is the real scope for x, y, z?

3. How the semantics can be correctly formalized? How it can be generally and consistently implemented?

4. How this situation can be correctly statically typed?

The ODMG OQL grammar introduces at least 7 different contexts where such auxiliary naming can be used. Till now, the formal semantics of all these contexts is non-existent (explained by toy examples rather than by a formal specification). An attempt to define the semantics of one of these contexts (on the level of static typing only) obviously failed, as shown above.

Indeed, SBQL is the first query language where the semantics of such auxiliary naming is generally and correctly defined. The semantic definition is very simple, but has no precedents in any mathematical theory, in programming languages and in former query languages. This idea was implemented in the Loqis system in 1988, at least five years earlier than the ODMG proposed the first version of the object-oriented database standard. Our solution of the problem requires the notion of binder (introduced previously) that occurs only in SBA and SBQL. The semantics covers static strong typing (as will be shown later) and is fully consistent with the introduced two stack apparatus. Moreover, it makes no special situation for query optimization methods.

In our attempts to formalize the semantics of auxiliary naming we have rejected the relational calculus as not sufficiently precise for all the contexts where such naming can occur. Our idea was to integrate smoothly auxiliary naming with the stack-based idea. The lighthouse that we would like to follow was the following:

Each name occurring in a query or in a program must be subordinated to the same scoping and binding rules that are accomplished by the environment stack mechanism. This also concerns all auxiliary names defined inside queries.

After taking this tenet, the questions that we would like to solve were the following:

- How definitions of auxiliary names have to be internally represented?
- Where the definitions have to be stored?
- How the definitions have to be used when a particular auxiliary name occurring in a query has to be bound?
- When the definition of an auxiliary name has to be removed, assuming it is no more valid?

There is a long history of several ideas that we tried to follow. In the Linda system we have assumed that the definition of an auxiliary name in a query is equivalent to the declaration of a local programming variable. Unfortunately, this was inconsistent; in particular, makes no answer to the last question.

The answer for all the above questions was invented in 1988, during the implementation of the Loqis system. The solution appears to be simple, universal (covering all the cases of auxiliary naming), very easy to implement, and free from disadvantages. The solution can be
very surprising for all the fans of the relational calculus, because it totally rejects the concept of “variable”.

Following OQL we assume the syntax:

query ::= query as name

As usual in SBQL, such a query has the semantics independent from other operators. It is as follows:

- Let query q returns bag\{a_1, a_2, a_3, ...\}. Then, the query q as n returns bag\{n(a_1), n(a_2), n(a_3), ...\}.
- Let query q returns sequence\{a_1, a_2, a_3, ...\}. Then, the query q as n returns sequence\{n(a_1), n(a_2), n(a_3), ...\}.
- Let query q returns an individual element a. Then, the query q as n returns n(a).

As previously, the operator may occur in any place of a query if it makes sense for the programmer and the strong typing is not violated. In particular, the operator can be nested. For instance, if q returns an individual element a, then (q as n_1) as n_2 returns a nested binder n_2(n_1(a)). As will be shown, nesting of this operator is important for creating complex values (nested structures). There is no constraint concerning the argument query q. For instance, 2 as two is a legal query that returns the binder two(2).

We will show in the following that this simple operator in connection with other SBQL operators presents a very powerful and universal facility to deal with almost all the contexts when auxiliary names for queries or sub-queries are needed. This concerns the following cases:

- “Iteration variables” (tuple, domain, collection “variables”) that occur e.g. in a from clause of SQL and OQL; for instance Dept as d. The programmer can use such a variable in any place where he/she wants to; unlike OQL and SQL, this is not limited to any construct of SBQL. In particular, the construct can be used to define dependent joins as in OQL, for instance, below is an SBQL query that is the equivalent of the above OQL query:

(Student as x join (x.takes.Lecture) as y join (y.taught_by.Professor) as z) where z.rank = "full professor"

In the query we have skipped some parentheses assuming that operator as is stronger than join, dot is stronger than where and =, and operators join are executed from left to right.

- “Variables bound by quantifiers”: as we will see, SBQL does not force to use such variables. If necessary, such a “variable” can be established by the operator as; for instance ∃ Emp as e (e.sal > 10000) .

- Labels of elements of a structure(s) created by a query, for instance,

Emp.( name as n, sal as s )

Such application of the operator can be used to construct a nested complex value. For instance, the query
("Lee" as name, 900 as sal, ("Rome" as city, "Boogie" as street, 13 as house) as address) as Emp

returns the nested binder

\[ Emp( \text{struct}\{ \text{name="Lee"}, \text{sal}(900), \text{address}(\text{struct}\{ \text{city="Rome"}, \text{street="Boogie"}, \text{house}(13) \}) \} ) \].

- "Cursor" in for each statements. As for quantifiers, such a cursor is not obligatory in SBQL (as will be shown later). If necessary, such a "cursor" can be established by the operator as; for instance,

\[ \text{for each } \text{Emp as e do } e.\text{sal} := e.\text{sal} + 100; \].

- Defining new attributes for virtual views; the examples will be shown later, when we will come to the theme.

- Some other SBQL contexts that so far have no counterparts in other query languages; for instance, overloading views.

One of the SBQL contexts of auxiliary naming is not covered by the operator as. Thus we have introduced in SBQL another naming operator that we denote group as. As before, group as is a unary algebraic operator parameterized by a name. The definition of the group as operator is even simpler that the definition of the as operator.

We assume the syntax:

\[ \text{query ::= query group as name} \]

As usual in SBQL, such a query has the semantics independent from other operators. Semantics is the following:

\[ \text{result}( q \text{ group as } n ) = n(\text{result}( q)) \]

The query \( q \text{ group as } n \) returns a single binder named \( n \), which wraps the entire result of the query \( q \). For instance, if \( \text{result}( \text{Emp} ) = \text{bag}\{i_1, i_5, i_9\} \) then \( \text{result}( \text{Emp group as } e ) = n(\text{bag}\{i_1, i_5, i_9\} ) \). The operator is necessary to define complex values that include named collections treated as single elements. It has also some meaning in conceptual modeling of queries, as it allows one to subdivide a complex query into several simpler queries (in a way that is different from the define clause of OQL). There is some conceptual intersection with the group by operator of SQL and OQL. In general, however, pragmatic contexts (practical applications) of using the group by operator and our group as operator are totally different.

As we noted before, in SBQL we do not introduce the group by operator because it is not necessary and causes far more problems than it is worth. An example of the use:

\[ (\text{Emp where } \text{sal} > 4000) \text{ group as } \text{FatGuys} . \]
\[ \text{(FatGuys where job = "programmer") group as } \text{FatProgs} , (\text{FatGuys where job = "manager") } \text{group as } \text{FatMgrs} . \]
\[ \text{bag}(\text{FatProgs}(. \text{ "Programmer " + name + " is well paid"}), \text{FatMgrs}(. \text{ "Manager " + name + " has good salary")}) \]

### 4.4 Non-Algebraic Operators

The essence of the Stack-Based Approach and SBQL is the concept of non-algebraic operators. We distinguish several non-algebraic operators: selection, projection/navigation, dependent/navigational join, quantifiers, ordering and (several kinds of) transitive closures. All non-algebraic operators are binary, i.e. they connect two queries. Despite syntactic...
similarity to algebraic operators, their semantics cannot be reduced to some algebra similar to the relational algebra or object algebras.

This assertion may look strange and unbelievably for the readers who have been convinced for decades that selection, projection and join are ordinary algebraic operators in the relational algebra. Such a treatment of these operators, however, was possible under severe constraints on their power and by assuming some not quite fair formal trick that affects their semantics. Our conclusion is even stronger: this unfair formal trick has caused that the research on query languages has gone into wrong direction and has severely impeded the progress in the domain.

The trick consists in some mix of a formal theory and its meta-language. Consider the expression of the relational algebra:

$$\sigma_{sal>1000}(Emp)$$

It consists of the selection operator $\sigma$ qualified by the predicate $sal > 1000$. Strictly speaking, the selection operator is not a single operator, but the infinite family of operators having as many elements as selection predicates. The expression $sal > 1000$ does not belong to the language of the algebra. This is a meta-language expression determining which selection operator from the infinite family has to be chosen. This expression is informal, is out of mathematics, essentially this is informal comment that is used to explain the topic.

Hence, at least half of the expression above is fake mathematics. The operator $\sigma$ and name $Emp$ are first-class citizens, they are a part of the mathematical theory. The attribute name $sal$, operator $<$ and constant 1000 are second-class citizens, they are a part of the informal meta-language, essentially a comment. Semantics of the second part cannot be expressed in terms of the relational theory and there is no theory which deals with it. Of course, we know the argument that the meta-language could be formalized too. But in this way we would obtain a new formal system that would be quite different from the relational algebra. (This can be done in many ways and SBA is actually one of them.) In the pure relational algebra even the natural join operator cannot be expressed, because names of attributes are not properties of the theory. Of course, we also know the argument that indices are “so simple” that there is no room for ambiguities. We reject this argument because in mathematics nothing is obvious, everything must come from axioms and every even smallest formal semantic problem is a big problem.

This especially concerns so-called “object-algebras”, where simple indices may take the form as in the above expression:

$$\sigma_{NetSal(sal +100)>1000}(Emp)$$

The index takes a complex form, with own semantics and pragmatics, but everything is fake mathematics: indices are comments rather than formulas.

A similar conclusion holds for all the other operators of the relational algebra. The above arguments can be repeated for the relational calculus and approaches based on formal logic. All these theoretical concepts subdivide the semantics of query languages into two worlds: (1) the semantics with strong mathematical treatment, and (2) the semantics which can be explained by informal comments only. If formal semantics of a query language such as SQL (claimed to be based on the relational algebra or relational calculus) concerns only a part of this language, the semantics of the entire query language is informal. Taking the current state of SQL, we can estimate that at most 10% of its functionality can be explained formally by the relational algebra or calculus. The next 90% is outside of any theory.
This subdivision of the concepts into formal and informal we have considered frustrating and unacceptable, especially for more sophisticated data models such as object-oriented and XML-oriented models. Moreover, the subdivision is completely unnecessary. It is caused by some religious attitude to the relational algebra and other lame database theories. It is not a big problem to develop another mathematical theory that consistently covers all the issues. The denotational semantics of programming languages is an excellent example of such a theory.

But actually we have lost our belief that the mathematical method is good for specification of any practical language. We have already presented the fundamental disadvantages of mathematics as a method of specification of semantics of practical artifacts. We follow our formal specification method based on operational semantics or abstract implementation. Our basic assumptions essentially distinguish our approach from the relational algebra and its object-oriented counterparts:

| In SBQL no operator is indexed by an informal meta-language expression. |
| In SBQL each data name occurring in a query has the same semantic treatment. |
| There is no subdivision on first-class and second class names or operators. |

In this way we avoid the subdivision of the SBQL semantics into formal and informal parts. All operators, including all algebraic operators such as + and <, and operators such as selection, projection and join, are on the same level of the semantic description. We also avoid subdivision of names occurring in queries into first class and second class citizens (as in the above relational algebra example). The object relativism principle requires the uniform treatment of object names occurring on different levels of object hierarchy. Moreover, from the very beginning we have assumed that each name occurring in a query is bound to run-time entities according to the same simple stack-based binding mechanism that we have explained previously.

Although the description presented below is simple and intuitive, it requires some attention and imagination from the reader, especially if he/she has little experience with recursive functions and compiler construction. We believe that understanding the mechanism that we propose will open for the Reader a quite new and very attractive world.

4.4.1 Procedure eval for Non-Algebraic Operators

All non-algebraic operators are defined according to the same semantic pattern. It is a bit surprising that apparently very different operators, such as selection, projection/navigation, join and quantifiers, have the same semantic core. It is, however, a very positive property allowing for many inferences independent on the kind of operator, including definition of semantics and some methods of query optimization.

Let $\theta$ be a symbol denoting a non-algebraic operator, and let $q_1 \theta q_2$ be a query with the operator $\theta$. The idea of the semantics is that the query $q_2$ is evaluated for every element $e$ of the bag (or sequence) returned by the query $q_1$. Each evaluation of $q_2$ is performed in a bit different run-time environment. Namely, if $q_2$ is evaluated for $e$, then the environment stack ENVS is augmented by the result of function nested applied to $e$. After evaluation of $q_2$ for this $e$ ENVS is reduced to the previous state. Assume that $\text{result}(q_1) = \text{bag}\{e_1, e_2, e_3\}$. Fig.4.4 presents successive states of ENVS. Query $q_2$ is evaluated three times. The final result of evaluation of $q_1 \theta q_2$ is a function of the result returned by $q_1$ and all the results returned by $q_2$; the function depends on $\theta$. 
Fig. 4.4 presents sequential processing of the elements of \( \text{bag}\{ e_1, e_2, e_3 \} \), but theoretically this can be done in parallel. However, parallel execution leads to hard issues that will not be considered in this description. Informally we can describe this procedure in the following terms:

1. Evaluate query \( q_1 \).
2. For each \( e \in \text{result}(q_1) \) do the following steps:
   - Calculate \( \text{nested}(e) \). It returns a set of binders.
   - Push \( \text{nested}(e) \) as the top section on ENVS.
   - Evaluate query \( q_2 \) in this new environment.
   - Calculate a partial query result through some function \( \text{partialResultOf}_\theta(e, \text{result}(q_2)) \); the function depends on \( \theta \).
   - Pop (remove) the top section from ENVS.
3. Merge all partial result into the final result. It is done by some function \( \text{mergePartialResults}_\theta(\text{partialResult}_1, \text{partialResult}_2, \ldots, \text{partialResult}_k) \), which also depends on \( \theta \).

If query \( q_1 \) returns a sequence, the procedure is similar, but the order of partial results is significant and is taken into account by the \( \text{mergePartialResults}_\theta \) function. The non-algebraic operators can also act on individual values. It is treated as a one-element bag, but for some operators (e.g. where, dot) if the result of the entire query will also be a one-element bag, then it is coerced to its element.

A part of the procedure \( \text{eval} \) that specifies the formal semantics of non-algebraic operators is presented below. Then, the specification will be explained in detail for particular non-algebraic operators.
procedure eval( q : query )
begin
    .......
    case q is recognized as $q_1 \theta q_2$: // the query q consists of a non-algebraic operator $\theta$
joining subqueries $q_1$ and $q_2$
    begin
        partialResults: bag of Result;
        partialResult, finalResult, e: Result;
        partialResults := bag{}; //empty bag
        eval( q_1 ); //Evaluation of the first query; the result bag is at the top of QRES
        for each e belonging to the bag at top( QRES ) do
            begin
                push( ENVS, nested( e ) ); //new section at ENVS
                eval( q_2 ); //Evaluation of the second query; the result is at the top of QRES
                partialResult := partialResultOf$_{\theta}$( e, top( QRES ) );
                partialResults := partialResults $\cup$ { partialResult };
                // new partialResult included to the partialResults bag
                pop( QRES ); //removing the result( q_2 ) from QRES
                pop( ENVS ); //removing the top section with nested( e ) from ENVS
            end;
        finalResult := mergePartialResults$_{\theta}$ ( partialResults ); // forming the final result
        pop( QRES ); //removing the result( q_1 ) from QRES
        push( QRES, finalResult );
        //the final result of the entire query is pushed at the top of QRES
    end
    .......
end

As can be deduced from this procedure, the state of ENVS is changing during evaluation, but
the final state is the same as the initial state. QRES is also used, but finally it is augmented by
one section keeping the entire result of $q_1 \theta q_2$.

Below we present descriptions of particular non-algebraic operators, including description of
corresponding functions partialResultOf$_{\theta}$ and mergePartialResults$_{\theta}$. We also present some
minimal changes to the procedure that may be required if the first query will return a
sequence or an individual element. In this chapter we do not present transitive closure
operators - they will be presented later.

4.4.2 Selection

Syntax: query ::= $q_1$ where $q_2$

Typing constraint: $q_2$ must return true or false.

Function partialResultOf$_{where}$: for given $e$ it returns bag{ e } if $q_2$ returns true, and bag{} (empty bag) if $q_2$ returns false.

Function mergePartialResults$_{where}$: it returns a bag – the sum of partial results.

For sequences: partial results keep order and the final result keeps the order too.
For individual elements: an element \( x \) is coerced to the bag \( \{ x \} \). The final result is thus \( \{ x \} \) if \( q_2 \) returns true, and \( \{ \} \) if \( q_2 \) returns false. Other operators can coerce final \( \{ x \} \) back to \( x \).

Examples (c.f. Fig.2.1 and Fig.2.2):

| Get references to employees earning more than 1000 | \( \text{Emp where } (\text{sal} > 1000) \) |
| Get a reference to the employee named “Poe”; the reference should be named \( x \) | \( (\text{Emp as } x) \text{ where } ((x\text{.name}) = \text{“Poe”}) \) |
| Get a reference to the Trade department located in Paris | \( (\text{Dept where } (\text{dname} = \text{“Trade”}) \text{ where } (\text{location contains} \text{ “Paris”}) \) |
| Get reference to the Trade department located in Paris (another formulation) | \( \text{Dept where } ((\text{dname} = \text{“Trade”}) \text{ and } (\text{location contains “Paris”}) \) |

More examples will be presented after introducing next operators. In further examples we will skip a lot of parentheses assuming some obvious (and a bit random) precedence rules of operators. In the concrete syntax these precedence rules must be of course formalized.

Fig.4.5 shows steps of the evaluation of a simple query (cf. Fig.2.2). Magenta boxes show the state of the environment stack. Grey boxes present results of particular (sub)queries.

![Fig.4.5. Steps of evaluation of the query Emp where sal > 1000](image)

### 4.4.3 Projection, Navigation and Path Expressions

**Syntax:**  \( \text{query ::= } q_1 \cdot q_2 \)

**Function** \( \text{partialResultOf}_e \): for given \( e \) it returns \( \text{top}(\text{QRES}) \), i.e. the result \( q_2 \).
Function `mergePartialResult`: it returns the bag – sum of partial results. Partial results cannot be sequences.

For sequences: partial results must be sequences. They are concatenated in the proper order into the final result.

For individual elements: if \( q_2 \) returns an individual element too, then the final result is \( \text{result}(q_2) \). Otherwise \( \text{result}(q_1) \) is coerced to a bag and the situation is as previous.

Examples (c.f. Fig.2.1 and Fig.2.2):

| Get references to salaries of all employees | \( \text{Emp} \cdot \text{sal} \) |
| Get references to salaries of employees earning more than 1000 | \( \text{Emp where} (\text{sal} > 1000) \cdot \text{name} \) |
| Get references to names of employees working in the Trade department | \( (((\text{Dept where} \ (\text{dname} = \text{"Trade"})) \cdot \text{employs}) \cdot \text{Emp}) \cdot \text{name} \) |
| Get references to names of employees working in the Trade department (path expression) | \( (\text{Dept where} \ (\text{dname} = \text{"Trade"})) \cdot \text{employs}.\text{Emp} \cdot \text{name} \) |
| Get references to worksIn pointers for employees earning more than 1000 | \( \text{Emp where} (\text{sal} > 1000) \cdot \text{worksIn} \) |
| Get references to worksIn pointers for employees earning more than 1000 | \( (((\text{Emp where} \ (\text{sal} > 1000)) \cdot \text{worksIn}) \cdot \text{Dept} \) |

Third and fourth queries show that path expressions are not a special construct in SBQL. A path expression is a combination of several binary dot operators. In the fourth example we have simply assumed the syntactic convention that dot operators are evaluated from left to right. Because dot operators can be freely combined with other operators, our path expressions are the most universal and orthogonal to other operators as one can imagine.

Note the difference between the two last queries. The first one identifies pointers \( \text{worksIn} \), and the second one identifies objects that the pointers are pointing to. In ODMG OQL and OMG OCL there is no possibility to express such a distinction. However, it is important for two reasons:

1. **Conceptual modeling**: a path expression clearly shows for the programmer where the navigation ends up;

2. **Updating operations**: sometimes we need to update pointers, and sometimes we need to update the objects that the pointers point to. No such distinction in OQL causes that one of these updating kinds cannot be expressed through the query language.

In ODMG OQL path expressions are limited by the assertion that the query before a dot (i.e. \( q_1 \) in our case) must return an individual element rather than a collection. If it returns a collection, then instead of \( q_1 \cdot q_2 \) one shall write `select q_2 from q_1`. In SBQL there is no such an assumption: the dot operator can act on individual elements and on collections as well. This feature of OQL looks unreasonable and causes our suspicion that the ODMG guys tend to confuse syntax and semantics.
4.4.4 Dependent/Navigational Join

Syntax: \( \text{query} := q_1 \text{ join } q_2 \)

Function partialResultOf join: for given \( e \) and \( \text{result}(q_2) = \text{bag}\{ r_1, r_2, \ldots, r_k \} \) the function returns \( \text{bag}\{ \text{struct}\{ e, r_1 \}, \text{struct}\{ e, r_2 \}, \ldots, \text{struct}\{ e, r_k \} \} \). The \( \text{bag}\{ r_1, r_2, \ldots, r_k \} \) is stored at the top of QRES. Results \( r_1, r_2, \ldots, r_k \) can be simple elements or structures. Note that we have assumed no directly nested structures, for instance, \( \text{struct}\{ a, \text{struct}\{ b, c \}, d \} \) is equivalent to \( \text{struct}\{ a, b, c, d \} \). Structures, however, can be nested assuming other query language constructors, for instance, \( \text{struct}\{ a, n(\text{struct}\{ b, c \}), d \} \) is not equivalent to \( \text{struct}\{ a, b, c, d \} \).

Function mergePartialResult join: it returns the bag-sum of partial results. Partial results cannot be sequences.

For sequences: partial results must be sequences of structures. They are concatenated in the proper order into the final result.

For individual elements: if \( q_1 \) and \( q_2 \) returns individual elements, then the final result is \( \text{struct}\{ e, \text{result}(q_2) \} \). Otherwise \( \text{result}(q_1) \) is coerced to bag and the situation is as previous.

Examples (c.f. Fig.2.1 and Fig.2.2):

| Get references to all employees with references to their salaries | Emp join sal |
| Get references to all employees with references to their worksIn pointer objects | Emp join worksIn |
| Get references to all employees with references to their departments | Emp join (worksIn.Dept) |
| Get references to all employees (named \( e \)) with references to their departments (named \( d \)) | (Emp as \( e \)) join ((e.worksIn.Dept) as \( d \)) |
| Assume that the Dept objects contain boss pointer objects leading to employees being managers of departments. Get references to names of employees working in Roma departments, together with references to names of their managers | ((Emp as \( e \)) join ((e.worksIn.Dept) as \( d \)), (e.name, d.boss.Emp.name) |

4.4.5 Quantifiers

Syntax: for quantifiers we use the traditional prefix syntax:

\[ \text{query} := \exists q_1 \text{ q}_2 | \forall q_1 \text{ q}_2 \]

Sometimes we will also use our standard infix syntax:

\[ \text{query} := q_1 \exists q_2 | q_1 \forall q_2 \]

In implementation symbols \( \exists \)and \( \forall \)should be substituted by keywords, so we assume also the syntax:

\[ \text{query} := \text{exists } q_1 \text{ such that } q_2 | \text{for any } q_1 \text{ holds } q_2 \]

Typing constraint: \( q_2 \) must return true or false.
Function partialResultOf∃ and partialResultOf∀: for given e it returns top(QRES), i.e. the result(q2).

Function mergePartialResult∃: it returns true if the bag of partial results contains at least one result true. Otherwise it returns false.

Function mergePartialResult∀: it returns true if the bag of partial results is empty or if all partial results are true. Otherwise it returns false.

For sequences: as for bags.

For individual elements: result(q1) is coerced to a bag and the situation is as previous.

Examples (c.f. Fig.2.1 and Fig.2.2):

| Is it true that some employee earns less than 1000? | ∃Emp (sal < 1000) 
Emp ∃ (sal < 1000) |
| --- | --- |
| Get departments where all the employees earn more than 1000 | Dept where (∀(employs.Emp)(sal > 1000)) 
Dept where ((employs.Emp) ∀ (sal > 1000)) |
| Is it true that there are employees with no address? | ∃(Emp as e ) count(e.address) = 0 |
| Get departments where all employees have an address | Dept where (∀ (employs.Emp) exists(address)) |
| Assume that the Dept objects contain boss pointer objects leading to employees being managers of departments. Get departments having employees earning more than their bosses. | (Dept as d) where ∃((d.employs.Emp) as e) 
(e.sal > d.boss.Emp.sal ) |
| Is it true that there is a department where all the employees for sure live in Paris? | ∃ Dept ( ∀ (employs.Emp) ( ∃ address (city = “Paris”))) |
| Is it true that there is a department where all the employees live in Paris or their address is unknown? | ∃ Dept ( ∀ (employs.Emp) ( ∀ address (city = “Paris”))) |

More detailed examples of SBQL queries and comparisons with SQL and OQL can be found in the next section.

4.5 SBQL Examples and Comparisons

In Fig.4.6 we present a relational and an equivalent object-oriented schema that will be used in our examples addressing the AS0 model. The schemata are basically self-explanatory. In the relational schema arrows with dashed lines denote primary-foreign keys dependencies; an arrow is leading from a foreign key to the corresponding primary key. The object-oriented schema is presented in the UML-like syntactic fashion, with two differences. First, cardinalities are put after each defined entity (default cardinality [1..1] is omitted). For instance, cardinalities [0..1] after Address denote that for a particular employee the address can occur or not. Note that such information cannot be expressed in the relational schema. A department can be located in one or more locations (this information also cannot be expressed...
in the relational schema). Second, we represent AS0 pointer objects associated with particular objects, thus for instance, worksIn pointer objects are close to Emp objects. In UML the worksIn role of the association would be close to the Dept class, just otherwise to our convention. Our examples will address the object-oriented schema. If we will address the relational schema we say that explicitly.

![Relational schema](image)

**Object-oriented schema** (class diagram)

![Object-oriented schema](image)

Fig. 4.6. Relational schema and an equivalent object schema used in examples

| E.4.1 | Get full information on employees. |
| SBQL: | Emp |
| SQL: | select * from Emp |
| OQL: | Emp |

We consider the SQL syntactic sugar unnecessary and contradictory to the tradition of programming languages. “User-friendliness” of this sugar is a false stereotype, especially for complex nested queries. Note also the semantic difference: the SBQL query returns a bag of references to all Emp objects, while the SQL query returns the entire Emp table. The OQL query returns in this case the set of Emp objects; a reference to an object is not a feature of the
ODMG standard. SBQL queries never return objects but references to objects, perhaps within some more complex structures.

### E.4.2 Get names of all the employees.

**SBQL:** \(Emp\).name

In SQL (see the relational schema) and in OQL this query can be formulated as:

**SQL:**

```
select name from Emp
```

**OQL:**

The SBQL query returns a bag of references to all name subobjects within Emp objects, while the SQL query returns an one-column table with values of the names. The OQL query returns a bag of literals, i.e. strings. OQL does not allow for the query \(Emp\).name; the explanation of this restriction is at least strange. Because the SBQL in this case returns references to name subobjects, it is much more universal, for instance, can be an argument of updating statements. SQL and OQL by definition reject such a possibility.

### E.4.3 Get all the information on the employee named “Doe”.

**SBQL:** \(Emp\) where name = “Doe”

In SQL (see the relational schema) and in OQL this query can be formulated as:

**SQL:**

```
select * from Emp where name = “Doe”
```

**OQL:**

The SBQL query returns a bag of references to the Emp objects storing the information about employees named Doe. The references can be consumed by another SBQL operator or by a programming statement based on SBQL, for instance, by the dot operator or by update statement. If the bag consists of exactly one element, it can be automatically coerced (by another operator) to this element. The SQL query returns a table with rows containing the values of Doe tuples. The OQL query returns a bag with Doe objects.

### E.4.4 Get the salary of “Doe”.

**SBQL:** \((Emp\) where name = “Doe”).sal

In SQL (see the relational schema) and in OQL this query can be formulated as:

**SQL:**

```
select sal from Emp where name = “Doe”
```

**OQL:**

The SBQL query returns a bag of references to the sal objects that are within the Emp objects with name equal to “Doe”. The SQL query returns one-column table with sal values of Doe tuples. The OQL query returns a bag of literals - values of sal of Doe objects. Due to the fact that SBQL returns references to sal objects rather than their values (as SQL and OQL do) the above query can be used in an updating statement (set Doe’s salary to 5000):

**SBQL:** \((Emp\) where name = “Doe”).sal := 5000;

This is equivalent to the SQL updating statement:

**SQL:**

```
update Emp set sal = 5000 where name = “Doe”
```
However, SQL makes no explicit relationships of this statement with the above SQL query, what is violation of the orthogonality principle, resulting in many further disadvantages, in particular implementation effort overhead, lower potential for query optimization, longer user documentation, etc. Note also a small (but painful) syntactic fault: the operator = in one SQL statement is used in two meanings (assignment and comparison) what could be an unexpected difficulty for beginners.

In the following we return to the topic of nesting SBQL queries into imperative statements.

### E.4.5

<table>
<thead>
<tr>
<th>SBQL:</th>
<th>Get e#, name and job of employees earning more than 1000.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{SBQL: } (\text{Emp where sal} = 1000).\text{struct}(e#, \text{name, job}) )</td>
<td>( \text{SBQL: } (\text{Emp where sal} = 1000).\text{(e#, name, job)} )</td>
</tr>
</tbody>
</table>

In SQL (see the relational schema) and in OQL this query can be formulated as:

<table>
<thead>
<tr>
<th>SQL:</th>
<th>OQL:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>select e#, name, job from Emp where sal = 1000</code></td>
<td><code>select e#, name, job from Emp where sal = 1000</code></td>
</tr>
</tbody>
</table>

The SBQL query returns a bag of structures `struct\{i\_e\#, i\_name, i\_job\}` with references to proper subobjects stored within a corresponding `Emp` object. Note that the elements of the structure are unnamed (although names can be derived from names of corresponding objects). In general (in contrast to C/C++ and many other languages), for conceptual closure SBQL takes the point of view that structures may contain unnamed elements. References returned by the query are more universal than values returned by SQL and OQL queries, because references can be automatically dereferenced by some operators (as will be shown later), but also can be used within updating statements. This is impossible in SQL and OQL.

### E.4.6

<table>
<thead>
<tr>
<th>SBQL:</th>
<th>Get worksIn references for employees named Doe.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{SBQL: } (\text{Emp where name} = \text{“Doe”}).\text{worksIn} )</td>
<td>( \text{SBQL: } (\text{Emp where name} = \text{“Doe”}).\text{worksIn} := (\text{Dept where dname} = \text{“Sales”}); )</td>
</tr>
</tbody>
</table>

For SQL this query is irrelevant. In OQL such a query cannot be expressed. Such a query however makes a sense because it can be a part of more complex queries (see the next one). Such a query makes also a sense because a `worksIn` pointer needs to be updated. This updating feature is commonly neglected by many proponents of query languages. For instance, to move Doe to the `Sales` department it is necessary to assign to this pointer the proper reference. In the imperative extension of SBQL such a request can take the form of the following statement:

On the left side of this statement we receive a reference to a `worksIn` pointer. On the right side of the statement we obtain a reference to the `Sales` department, which has to be assigned to the pointer. Such an assignment can automatically trigger an action on the twin `employs` pointer, as in the C++ and Java bindings of the ODMG standards. Note that the distinction between assigning a reference and assigning an object is resolved by the strong typing system. In the above case the `worksIn` pointer object is typed as a relationship, hence it can be automatically inferred that the assignment concerns a reference to the `Dept` object rather than the `Dept` object itself. The SBQL strong typing system is explained in [Hryn05, Lent06b, ODRA10, Sten06, Subi04].
E.4.7 Get all information on departments for employees named Doe.

**SBQL:**

```sql
((Emp where (name = “Doe”).worksIn).Dept
```

**SBQL:**

```sql
(Emp where name = “Doe”).worksIn.Dept
```

The query returns a bag of references to *Dept* objects for persons named Doe. In SQL (see the relational schema) this query can be formulated as follows (one of several possible ways):

**SQL:**

```sql
select * from Dept where d# in select d# from Emp where name = “Doe”
```

In OQL this query can be formulated as (one of several possible ways):

**OQL:**

```sql
select d from Emp as e, e.worksIn as d where e.name = “Doe”
```

Note that the OQL query avoids using the name *Dept*, which seems to be an advantage: queries are shorter. We consider such a feature as disadvantageous for two reasons:

1. For updating there is necessity to distinguish a reference to *worksIn* pointer and a reference to a *Dept* objects. OQL makes such a distinction impossible, thus it would create difficulties if one would smoothly extend OQL with updating statements.

2. SBQL queries are more legible, because they explicitly show where the navigation is finished, i.e. on *Dept* objects in this case. In OQL the programmer sees only *worksIn* relationship thus must verify the schema (which could be very large) to be sure what will be the result of the navigation. Hence, the SBQL convention is better from the conceptual modeling point of view.

E.4.8 Get names of departments for employees named Doe.

**SBQL:**

```sql
(((Emp where (name = “Doe”).worksIn).Dept).dname
```

**SBQL:**

```sql
(Emp where name = “Doe”).worksIn.Dept.dname
```

The query returns a bag of references to *dname* objects for persons named Doe. In SQL (see the relational schema) this query can be formulated as follows (one of several possible ways):

**SQL:**

```sql
select dname from Dept where d# in select d# from Emp where name = “Doe”
```

In OQL this query can be formulated as (one of several possible ways):

**OQL:**

```sql
select e.worksIn.dname from Emp as e where e.name = “Doe”
```

The example illustrates so-called *path expressions*, that is, smooth navigation along some path in the database schema graph. In SBA and SBQL path expressions are side-effects of the definition of the binary dot operator (see the first query) and the syntactic convention assuming the evaluation from left to right (see the second SBQL query). Path expressions can be as long as necessary, for instance (get the name of the Doe’s boss):

**SBQL:**

```sql
(Emp where name = “Doe”).worksIn.Dept.boss.Emp.name
```

Because we do not define path expressions as special syntactic and semantic constructs (as for instance in [Kim89, Kife92, Chri96, Zani83, Subi83]) we receive full freedom in combining path expressions with other SBQL operators. We believe that our path expressions have the highest universality from all the proposals that can be found in the literature and in other proposals of query languages. For instance, the following SBQL query specifies names and cities for employees working in the department managed by Kim:

**SBQL:**

```sql
(Dept where (boss.Emp.name) = “Kim”).employs.Emp
```
(name, if exists(Address) then Address.city else “No address”)}

Other proposals concerning path expressions are usually informal concerning their semantics, or the semantic specification is limited. In particular, OQL allows for path expressions, but only in cases when a sub-query before a dot returns a single element rather than a collection. In other cases OQL requires to use the select...from...where syntax, as in the above OQL query. Such limitations and requirements are of course rejected during the development of SBQL.

**E.4.9** Get all information on employees earning more than Doe.

**SBQL:** \(\text{Emp where } \text{sal} > ((\text{Emp where name} = \text{“Doe”}).\text{sal})\)

The query illustrates the possibility to nest queries. It contains name \text{sal} two times, but due to the stack-based semantics each of them is bound differently: first \text{sal} is bound on ENVS having two sections, and the second \text{sal} is bound on ENVS having three sections. In SQL (see the relational schema) and OQL this query can be formulated as follows (one of several possible ways):

**SQL:** \(\text{select } * \text{ from Emp where sal} > \text{select sal from Emp where name} = \text{“Doe”}\)

**OQL:** \(\text{select struct(employee:e, department:d) from Emp as e, e.worksIn as d}\)

In our opinion, such SQL/OQL queries will also require some environment stack concept, but it is not explicit in the description of the languages’ semantics.

**E.4.10** For each employee return all information about the employee and his/her department.

**SBQL:** \(\text{Emp join (worksIn.Dept)}\)

The query uses the dependent join operator. The result is a bag of structures \(\text{struct}\{i_{\text{Emp}}, i_{\text{Dept}}\}\), where \(i_{\text{Emp}}\) is a reference to an \text{Emp} object and \(i_{\text{Dept}}\) is a reference to the \text{Dept} object associated with the \text{Emp} object by the \text{worksIn} pointer. The corresponding OQL query is the following:

**OQL:** \(\text{select struct(employee:e, department:d) from Emp as e, e.worksIn as d}\)

This is not a strict equivalent because in OQL elements of structures must have tags (as employee and department in the above query). Besides that it is not quite clear what such an OQL query will return - semantics of OQL is rather obscure (OQL does not introduce the concept of a reference) and this operator is improperly described as “cartesian product”. In SBQL such a query can also be issued to the relational structure:

**SBQL:** \(\text{Emp join (Dept where worksIn} \!= \!d\#\)\)

**SBQL:** \(\text{struct(Emp, Dept) where worksIn} \!= \!d\#\)

**SBQL:** \(\text{(Emp, Dept) where worksIn} \!= \!d\#\)

The last query has an equivalent in SQL:

**SQL:** \(\text{select } * \text{ from Emp, Dept where worksIn} \!= \!d\#\)

The equivalence is not strict, because the SQL query returns a table of values rather than references. In particular, such SBQL query can be used in updating statements, while the SQL query cannot.
For each department get all information together with the average salary of their employees.

**SBQL:**  
Dept join avg(employs.Emp.sal)

The result is a bag of structures `struct{iDept, average_salary}`, where `iDept` is a reference to a `Dept` object and `average_salary` is the real number being the average salary within this department. Changes of ENVS states during evaluation of this query are presented in Fig.4.7. An equivalent OQL query requires the use of the `group by` operator, but for obscure semantics of this feature we do not risk to express it. A similar SQL query also requires the use of the `group by` operator:

**SQL:**  
select d.*, avg(e.sal) from Dept as d, Emp as e where d.d# = e.worksIn group by d.d#

The disadvantage of this query is the necessity to use the `group by` operator, which is non-orthogonal to other operators, implies far context semantic dependencies (is not compositional), is difficult to optimize, causes some semantic reefs and sometimes requires an additional `having` clause. Note that the SBQL query has 10 lexical tokens, while the equivalent SQL query - 31 tokens. We comment this difference in the next section.

We also show that SBQL query can also address the relational schema, with no necessity to use the `group by` option:

**SBQL:**  
Dept join avg((Emp where worksIn = d#.sal))

**SBQL:**  
(Dept as d) join avg(((Emp as e) where e.worksIn = d.d#.e.sal))

Fig.4.7. ENVS during evaluation of the query `Dept join avg(employs.Emp.sal)`

This PowerPoint presentation shows all the steps that are necessary to evaluate this query for a tiny database.

Get the average number of employees in all the departments.
SBQL: \[
\text{avg}(\text{Dept}.\text{count}(\text{employs}))
\]

For the relational schema:

SBQL: \[
\text{avg}(\text{Dept}.\text{count}(\text{Emp} \text{where } d\# = \text{worksIn}))
\]

In the SQL-89 standard this query can be expressed through an additional view. In SQL-92 this query can be expressed by the `group by` clause. However, it leads to a well-known semantic reef, which concerns e.g. the case when some department has no employees. Such a department is not taken into account, hence the average will be different from expected. In SBQL such a semantic reef does not occur.

**E.4.13**

For each employee that earns more than 2000 and works in a department located in Paris get name, job, department name and the name of his/her boss.

SBQL: \[
((\text{Emp join (works_in.Dept)} \text{where } \text{sal} > 2000 \text{ and } \text{“Paris” in loc}). (\text{name, job, dname, (boss.Emp.name)}))
\]

In the result we obtain a bag of structures `struct\{iempname, iempjob, idname, ibossname\}`, where references `iempname`, `iempjob`, `idname`, `ibossname` refer to particular attributes. A similar SQL query is about two times longer:

SQL: \[
\begin{align*}
\text{select e.name, e.job, d.dname, b.name} \\
\text{from Emp as e, Dept as d, Emp as b, Location as l} \\
\text{where e.sal > 2000 and l.loc = “Paris”} \\
\text{and e.worksIn = d.d\# and d.boss = b.e\# and l.d\# = d.d\#}
\end{align*}
\]

**E.4.14**

Get departments employing a professional for any job in the company.

SBQL: \[
\text{Dept where ”distinct(Emp.job) as j ($employs.Emp (j = job))}
\]

A similar SQL query is very hard to express.

**E.4.15**

Is it true that each department employs an employee earning more than his/her boss?

SBQL: \[
\forall \text{Dept} (\exists \text{employs.Emp (sal > boss.Emp.sal))}
\]

SBQL: \[
\forall \text{Dept as d (\exists d.employs.Emp as e (e.sal > d.boss.Emp.sal))}
\]

Such a query cannot be expressed in SQL, because SQL does not introduce queries returning boolean values. Moreover, because SQL has no explicit quantifiers, a similar query (e.g. returning 1 for `true` and 0 for `false`) requires the use of `exists` or `count` functions, what leads to an extremely clumsy statement. In OQL the query can be expressed as follows:

OQL: \[
\begin{align*}
\text{for all } d \text{ in Dept : exists e in select x from d.worksIn as x :} \\
\text{e.sal > select y.sal from d.boss as y}
\end{align*}
\]

The reader may now evaluate if the `select...from...` sugar and the obligatory use of auxiliary iteration variables are useful or annoying features. Note that the first SBQL query has 17 lexical elements, while the OQL query - 31. Which of the queries is easier to write and comprehend?
E.4.16  Get names and salaries of designers who earn more than their bosses.

We present several styles that can be applied to formulate this SBQL query. First we present the SQL style (see the relational schema):

```sql
SBQL, SQL style: 
```((
Emp as e,
Dept as d,
Emp as b
where e.job = "designer" and e.worksIn = d.d#
and d.boss = b.e# and e.sal > b.sal
). (e.name, e.sal))
```

A corresponding SQL query is very similar; basically there are only minor syntactic differences:

```sql
SQL: 
select e.name, e.sal
from Emp as e, Dept as d, Emp as b
where e.job = "designer" and e.worksIn = d.d#
and d.boss = b.e# and e.sal > b.sal
```

We can also present this query in the OQL style:

```sql
SBQL, OQL style: 
```(((
Emp as e
join e.worksIn.Dept as d
join d.boss.Emp as b
where e.job = "designer" and e.sal > b.sal
). (e.name, e.sal))
```

A corresponding OQL query has also only minor syntactic differences:

```sql
OQL: 
select e.name, e.sal
from Emp as e, e.worksIn.Dept as d, d.boss.Emp as b
where e.job = "designer" and e.sal > b.sal
```

SBQL allows for more compact form of this query:

```sql
SBQL: 
(Emp where job = "designer" and sal > (worksIn.Dept.boss.Emp.sal)).
(name, sal)
```

SBQL allows also for the “domain calculus” style of this query (see the relational schema). This style is sometimes claimed to be more friendly and better to optimize (we do not believe in that). Sometimes OQL is claimed to be based on this style (although this is contradictory to another claim that OQL is similar to SQL, which is based on the “tuple calculus” style). The style is also the property of QBE and RDQL.

```sql
SBQL, domain calculus style: 
(((Emp.(name as en, sal as es, job as ej, worksln as ed)),
(Dept.(d# as dd, boss as db)), (Emp.(sal as bs, e# as be)))
where ej = "designer" and ed = dd and db = be and es > bs). (en, es)
```

By these examples we would like to show that SBQL allows for many styles of querying on equal rights. The old debates on advantage of one style over another one are totally irrelevant on the ground of SBA and SBQL. Any inferences concerning ability of particular database models to support query languages are attempts to make false stereotypes. Contrary to the Michael Stonebraker’s thesis [Ston00] that relational and object-relational database models
have unique possibilities to build query languages, we argue that all database models have similar capabilities concerning query languages. There is no advantage of a particular model in this respect, including the relational and object-relational models.

4.5.1 Comparison of Queries in SBQL and LINQ

LINQ (Language INtegrated Queries) is a product of the Microsoft Research [Linq07, Linq10] that is developed within the .Net platform. It can be integrated with several programming languages developed and implemented for .Net, in particular, for C#. LINQ is integrated with C# rather than embedded, because LINQ queries follow the same strong typing system as for C#. Syntactically LINQ queries are not considered strings within C#, but a kind of expressions. However, in contrast to SBQL, LINQ queries and C# expressions are different syntactic categories. The comparison shows some flavor of both languages and allows for the reader to feel their advantages and disadvantages. We believe that LINQ has no advantages over SBQL, its queries are frequently rather awkward and the expressive power of LINQ is much lower than the power of SBQL.

The comparison is performed on the basis of the database schema presented in Fig.4.8. An XML file presented in Fig.4.9 is an example data fulfilled according to this schema. Such a file can be mapped to the SBQL AS0 model in the ODR A system. Attributes id and idref are mapped as bidirectional relationships (pointer objects).

![Fig.4.8. Schema used in examples](image)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<deptemp>
  <Emp id="i1">
    <name>Doe</name>
    <sal>2500</sal>
    <worksIn idref="i17"></worksIn>
    <manages idref="i17"></manages>
  </Emp>
  <Emp id="i5">
    <name>Poe</name>
    <sal>2000</sal>
    <worksIn idref="i22"></worksIn>
  </Emp>
</deptemp>
```
We present rather typical queries. SBQL has operators such as transitive closures, fixed-point equations and operations on range numbers that are absent in SQL and in LINQ; thus we avoid presenting such queries that are impossible to express in LINQ by definition. SBQL is not only a query language, but a full object-oriented database programming language with all the well-known functionality (procedures, functions, classes, methods, etc.) and some less-
known functionality (updatable views, protocols for integrating distributed resources and others). Such functionalities are also not presented in this comparison.

### E.4.17
Get departments together with the average salaries of their employees:

<table>
<thead>
<tr>
<th>SBQL</th>
<th>LINQ</th>
</tr>
</thead>
</table>
| `Dept join avg(employs.Emp.sal)` | `var query1 = from d in Dept
select new
dpt = d,
    avg = (from e in d.employs
        select e.sal).Average();` |

LINQ forces the use of internal variables in queries (such as `d` and `e` in the above query). In SBQL, similarly to SQL, such variables can be avoided in majority of cases. Moreover, LINQ uses at least three different kinds of auxiliary naming: iteration variables (`d` and `e`), variables used in lambda notation, structure field labels (`dpt` and `avg`) and perhaps others. In SBQL there is only one semantic category of auxiliary naming that can be used in all contexts (iteration variables, variables bound by quantifiers, structure field labels, etc.). Unification of this auxiliary naming makes the language conceptually and semantically simpler, more orthogonal and better prepared for query optimization.

### E.4.18
Get name and department name for employees earning less than 2222:

<table>
<thead>
<tr>
<th>SBQL</th>
<th>LINQ</th>
</tr>
</thead>
</table>
| `(Emp where sal < 2222).(name, worksIn.Dept.dname)` | `var query2 = from e in Emp
where e.sal < 2222
    select new
    {EmpName = e.name,
        DeptName = e.worksIn.dname};` |

### E.4.19
Get names of employees working for the department managed by Bert:

<table>
<thead>
<tr>
<th>SBQL</th>
<th>LINQ</th>
</tr>
</thead>
</table>
| `(Emp where (worksIn.Dept.boss.Emp.name) = "Bert").name` | `var query3 = from e in Emp
where e.worksIn.boss.name == "Bert"
    select e.name;` |

The LINQ query avoids using `Dept` after `worksIn` and `Emp` after `boss`, which seems to be the advantage: queries are shorter. As we explained in the previous section, for two reasons we consider such a feature disadvantageous.
### E.4.20

**Get the name of Poe's boss:**

<table>
<thead>
<tr>
<th>SBQL</th>
<th>(Emp where name = &quot;Poe&quot;). worksIn.Dept.boss.Emp.name</th>
<th>Lexical units</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINQ</td>
<td>var query4 = from e in Emp where e.name == &quot;Poe&quot; select e.worksIn.boss.name;</td>
<td>18</td>
</tr>
</tbody>
</table>

The same remark as above.

### E.4.21

**Names and cities of employees working in departments managed by Bert:**

<table>
<thead>
<tr>
<th>SBQL</th>
<th>(Dept where (boss.Emp.name) = &quot;Bert&quot;), employs.Emp.(name, if exists(address) then address.city else &quot;No address&quot;)</th>
<th>Lexical units</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINQ</td>
<td>var query5 = from e in Emp where e.worksIn.boss.name == &quot;Bert&quot; select new { Ename = e.name, Ecity = (e.address == null ? &quot;No address&quot; : e.address.city); }</td>
<td>41</td>
</tr>
</tbody>
</table>

The same remark as above. Some professionals severely criticize nulls in databases [Date86b, Date86c, Date92b, Subi01b, Subi96, Subi98], due to inconsistencies that nulls lead to. See the chapter devoted to irregular data. SBQL does not introduce null values. Instead, it deals with (nested) collections having the cardinality [0..1].

### E.4.22

**Get the minimal, average and maximal number of employees in departments:**

<table>
<thead>
<tr>
<th>SBQL</th>
<th>(Dept.count(employs)) groupas counts. (min(counts), avg(counts), max(counts))</th>
<th>Lexical units</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINQ</td>
<td>var counts = from d in Dept select d.worksIn.Count(); var query6 = new { Min = counts.Min(), Avg = counts.Average(), Max = counts.Max() }</td>
<td>42</td>
</tr>
</tbody>
</table>

LINQ has subdivided a single query into two programming statements. This means shifting query processing to procedural capabilities, which is disadvantageous for conceptual modeling and for query optimizations. We are not sure what are possibilities of LINQ concerning nested queries and is it possible to develop efficient optimization methods for nested queries. SBQL nested queries are optimized by powerful methods.
### E.4.23
For each department get its name and the sum of salaries of employees being not bosses:

<table>
<thead>
<tr>
<th>SBQL</th>
<th>Dept.(dname, sum(employs.(Emp where not exists(manages)).sal))</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINQ</td>
<td>var query7 = from d in Dept</td>
</tr>
<tr>
<td></td>
<td>select new</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>DeptName = d.dname,</td>
</tr>
<tr>
<td></td>
<td>StaffSalary = (from e in d.employs</td>
</tr>
<tr>
<td></td>
<td>where e != d.boss</td>
</tr>
<tr>
<td></td>
<td>select e.sal).Sum()</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
</tbody>
</table>

| Lexical units | 22 |

### E.4.24
Is it true that each department employs an employee earning the same as his/her boss?

<table>
<thead>
<tr>
<th>SBQL</th>
<th>forall (Dept as d)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>forsome ((d.employs.Emp minus d.boss.Emp) as e)</td>
</tr>
<tr>
<td></td>
<td>(e.sal = d.boss.Emp.sal)</td>
</tr>
</tbody>
</table>

| LINQ     | var query8 = ( from d in Dept                                   |
|          | select (                                                       |
|          |   from e in d.employs                                          |
|          |     where e != d.boss && e.sal == d.boss.sal                    |
|          |       select e).Any()                                           |
|          | ).All(found => found);                                          |

| Lexical units | 37 |

In the above LINQ example quantifiers Any and All are non-intuitive and far from traditional mathematical and query notation. There is no explicit variable bound by quantifiers. The notation found => found is a syntactic overhead of the lambda notation that could be difficult to explain. In our opinion, lambda notation is too complicated for the average programmer. The genericity that it implies by higher-order functions in non-consumable in languages such as C#. Functional polymorphic languages such as ML, Scheme and Haskell do not enjoy commercial success, despite big effort of academic communities.

### E.4.25
For each employee get the message containing his/her name and the percent of the annual budget of his/her department that is consumed by his/her monthly salary:

<table>
<thead>
<tr>
<th>SBQL</th>
<th>Emp .(&quot;Employee &quot; + name + &quot; consumes &quot; +</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>((sal * 12 * 100)/(worksIn.Dept.budget)) +</td>
</tr>
<tr>
<td></td>
<td>&quot;.% of the &quot; +</td>
</tr>
<tr>
<td></td>
<td>worksIn.Dept.dname + &quot; department budget.&quot;)</td>
</tr>
</tbody>
</table>

| LINQ     | var query9 = from e in Emp                                           |
|          | select "Employee " + e.name + " consumes " + |
|          |   ((e.sal * 12 * 100) / e.worksIn.budget) + |
|          |   ".% of the " + e.worksIn.dname + |
|          |   " department budget.";                                           |

| Lexical units | 37 |

### E.4.26
Get cities hosting all departments:

| Lexical |  |
The use of SelectMany, Distinct, All and Contains operators in LINQ looks complex and unnatural. It may require extensive training from the user.

Summing up, in average the above LINQ queries are 42% longer than SBQL queries and are obviously much less intuitive and legible.

### 4.5.2 Why the `group by` Operator is Unnecessary in Object Query Languages

The operator `group by` occurs in SQL and is proposed in other query languages, in particular, in the object-oriented query language OQL (by ODMG). In relational query languages the operator allows one to formulate a lot of useful queries that cannot be formulated otherwise. In particular, these queries cannot be formulated in the relational algebra\(^7\), making more evidence that the “relational completeness” of query languages is only a pseudo-scientific buzzword, with no sense and meaning. The operator was especially useful in connection with aggregate functions, for example (c.f. Fig.4.6, the relational case):

<table>
<thead>
<tr>
<th>E.4.27</th>
<th>For each department get its dept number, the number of employees and the average salary of employees:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL:</td>
<td><code>select worksIn, count(*), avg(sal) from Emp group by worksIn</code></td>
</tr>
</tbody>
</table>

Semantics of this construct is very simple. The table `Emp` is horizontally subdivided into groups according to the same value of `worksIn`. Then, for each such group the `select` clause calculates the number of tuples `count(*)` and the average salary `avg(sal)`. The result is a table with three columns, where the first can be named `worksIn` and two next are unnamed.

In SQL the operator is extended by the `having` clause that is a conditional expression allowing one to filter proper groups. The `having` clause can coexist with the `where` clause, for instance:

<table>
<thead>
<tr>
<th>E.4.28</th>
<th>For each department having more than 50 employees get its dept number and the average salary:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL:</td>
<td><code>select worksIn, avg(sal) from Emp group by worksIn having count(*) &gt; 50</code></td>
</tr>
</tbody>
</table>

The operator is relatively simple for a single table. In case of several joined tables it is not easy to use:

| E.4.29 | For each department having more than 50 employees get its dept number, dept |

---

\(^7\) There are special relational algebras that include the `group by` operator. In our opinion they are too opportunistic to consider them as a serious theoretical support.
name and the average salary:

| SQL: | select worksIn, d.dname, avg(e.sal) from Emp e, Dept d where e.worksIn = d.d# group by e.worksIn, d.dname having count(e.*) > 50 |

There are some additional rules of using this operator, in particular, all attribute names that occur in the select clause outside aggregate functions must also occur in the `group by` clause. For large SQL queries such far context dependency compromises orthogonality and could be the reason of programmer’s bugs. The far context dependency makes also some problem for static strong type checking; however, current SQL has no such a feature.

Some professionals consider the `group by` operator as a consequence of the flat (unnested) nature of relational tables. The `group by` operator temporarily subdivides the table into sub-tables, but such an operation is impossible in the classical relational model. Hence, a nested relational model could change the nature or the necessity of this operator. It is also considered disadvantageous for performance, as its definition is imperative rather than declarative. Hence many optimization methods become invalid if this operator is used. Some new methods are developed, but in general, because the operator requires sorting, the methods are not always as efficient as required.

The operator also leads to the well known semantic reefs where the result calculated by the query processor is different from the result expected by the programmer. The first reef is connected with empty groups, which are not taken into account by the operator.

**E.4.30** Get the departments together with the number of their employees:

| SQL: | select worksIn, count(*) from Emp group by worksIn |

Assume that the company has three departments, first with 10 employees, second with 20 employees and third with no employees, and the programmer wants to calculate the average number of employees in the departments. Obviously, the average is 10, but if it would be calculated according to the result returned by the above SQL statement, the average is 15. The empty department is not taken into account. Another semantic reef is connected with null values. If in the `worksIn` attribute nulls are allowed, then the `group by` operator collects all nulls within an additional group. Hence, the number of departments is increased by one, one group has null as its department number. We can imagine how many difficult bugs such a feature can generate, especially in the maintenance phase, when the administrator is forced to add “null is allowed” for definitions of some columns in the relational tables. Checking all `group by` operators in all applications of all clients that work with this database would not be an easy job.

In SQL the `group by` operator is tightly associated with aggregate functions `sum`, `min`, `max`, `avg` and `count`. If the functions act on a table that is not subdivided by `group by` into subtables, then they act as usual. However, if the aggregate functions are put in the context of the `group by` operator, then they calculate proper values for groups rather than on an entire table. This is a kind of semantic schizophrenia that is disadvantageous from the point of view of the perception of programmers and obvious evidence of non-orthogonality of aggregate functions with other query operators.

Despite the above well recognized disadvantages, ODMG OQL has introduced the `group by` operator. Unfortunately, the syntax and especially semantics of this operator are not clear. It makes little sense to repeat the definitions and examples presented in the “standard”, as they are rather below the commonly accepted scientific and technical formal specification standards. It seems that this operator is introduced with no deeper motivation, as a clone of
the corresponding SQL operator motivated by the (false) statement that OQL is only a minor
extension of SQL.

During the development of SBA and SBQL we have tried to develop a “civilized” version of
the operator that would satisfy the following assumptions:

- It should be general, with semantically precise formal specification;
- It should address any object store model, in particular AS0 – AS3;
- It should be fully orthogonal to other query operators, in particular, to aggregate
functions and quantifiers such as count and exists;
- It should be free of the above mentioned semantic reeifs related to empty groups and
null values;
- It should not imply the necessity of two different conditional clauses;
- It should not imply far context dependencies, such as specific for SQL the dependency
between group by and select clauses;
- It should be easy for static strong typing;
- It should not be bound to physical implementation, follow the declarative style of
query languages and be optimizable by regular query rewriting methods.

Unfortunately, despite many investigations, trials and dozens of examples we did not find the
solution that would satisfy these requirements. It is possible that other researchers will find it,
but actually we have lost any hope. Our conclusion is that the group by operator, motivated
purely by physical implementation, makes no chances to be integrated with other query
operators in such a way that all assumptions presented above would be fully satisfied.

Our attempts to specify this operator have led us to the conclusion that for object-oriented
models the operator is unnecessary, redundant and awkward. It can be smoothly and directly
substituted by dependent (navigational) join, dot and other SBQL operators. As we have
observed at the beginning, the operator requires subdividing a table into subtables, i.e. it
implicitly assumes a relational model with nested tables. But our store models AS0-AS3
allow for arbitrary nesting of stored data structures. Nesting is also the assumption concerning
query results. Hence the conceptual motivation for the group by operator disappears. It is
unable to introduce to a query language an essential new quality. Moreover, a lot of complex
queries that in SQL require the group by operator become incredible simpler in SBQL,
without this operator.

Below we illustrate on examples how we can avoid the group by operator in SBQL (c.f.
Fig.4.6, the object-oriented case).

| E.4.31 | (Compare E.4.17) For each department get its dept number, the number of
employees and the average salary of employees: |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>Dept.(d#, count(employs), avg(employs.Emp.sal))</td>
</tr>
</tbody>
</table>

| E.4.32 | (Compare E.4.18) For each department having more than 50 employees get its
department number and the average salary: |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>(Dept where count(employs) &gt; 50). (d#, avg(employs.Emp.sal))</td>
</tr>
</tbody>
</table>
### E.4.33

(Compare E.4.19) For each department having more than 50 employees get its dept number, dept name and the average salary:

**SBQL:**

\[
(\text{Dept }\text{where count(employs) > 50}).
\]
\[
(\text{d#}, \text{dname, avg(employs.Emp.sal)})
\]

Note that having clauses are not necessary.

### E.4.34

(Compare E.4.20) Get the average number of employees in departments:

**SBQL:**

\[
\text{avg( Dept.count( employs ) )}
\]

Note that unlike SQL there is no danger of the semantic reef causing omitting empty departments. All the departments having no employees will return \text{count(employs)} = 0, hence the average number of employees in departments is calculated correctly. To deliver the average number of employees in non-empty departments one can write:

### E.4.35

Get the average number of employees in non-empty departments:

**SBQL:**

\[
\text{avg(( Dept where exists( employs ))}.\text{count( employs )})
\]

SBQL avoids also the semantic reeves related to null values, because in SBQL null values are not supported. According to [Date86] and [Subi98] no consistent definition of null values is possible. Instead, in SBQL a null value is represented as a collection with the cardinality [0..1].

In SBQL we can also address relational databases. Below we show that all the above queries that require the group by operator in SQL can be formulated in SBQL without this operator. The prescription for such SBQL queries is very simple:

- Establish the set of values that are to be used for distinguishing groups and name it using the operator as.
- Use this name in an outer query to accomplish the required goal.

Sometimes naming suggested in the first step can be omitted. Below we present examples that follow this prescription (c.f. Fig.4.6, the relational case):

### E.4.36

For each department get its dept number, the number of employees and the average salary:

**SBQL:**

\[
\text{Dept.(d#, count(Emp where worksIn = d#),}
\]
\[
\text{avg((Emp where worksIn = d#).sal))}
\]

**SBQL:**

\[
\text{Dept join ((Emp where worksIn = d#) group as e).}
\]
\[
(d#, \text{count(e), avg(e.sal)})
\]

**SBQL:**

\[
(\text{Dept.d# as d})\text{ join ((Emp where worksIn = d) group as e).}
\]
\[
(d, \text{count(e), avg(e.sal)})
\]

### E.4.37

For each department having more than 50 employees get its dept number and the average salary:

**SBQL:**

\[
((\text{Dept as d})\text{ join ((Emp where worksIn = d.d#) group as e})
\]
\[
\text{where count(e) > 50}. (d.d#, \text{avg(e.sal)})
\]

### E.4.38

For each department having more than 50 employees get its dept number, dept
name and the average salary:

SBQL:

\[
((\text{Dept as } d) \join ((\text{Emp where } \text{worksIn} = d.d\#) \text{ group as } e ) \where \text{count}(e) > 50) \cdot (d.d\#, d.dname, \text{avg}(e.sal))
\]

Note that in this case (in contrast to SQL) including \textit{dname} into the final result has required a small obvious change only.

E.4.39 Get the average number of employees in departments:

SBQL:

\[
\text{avg}(\text{Dept.count}(\text{Emp d\# = worksIn}))
\]

Note that in this case (unlike SQL) no semantic reef is occurring: empty departments are taken into account during calculation of the average. One can also create the semantic equivalent of E.GB.4, but this must be explicitly determined by the programmer:

E.4.40 Get the average number of employees in non-empty departments:

SBQL:

\[
\text{avg}(\text{Dept join count}(\text{Emp d\# = worksIn} \text{ as } c \where c > 0).c)
\]

Some queries in SQL that require the \textit{group by} operator are difficult to formulate. Consider, for instance, the following SBQL query:

E.4.41 For each department having more than 50 employees get its dept number, dept name and the average salary of its clerks:

SBQL:

\[
((\text{Dept where } \text{count}(\text{employs}) > 50),
(\text{d\#, dname, avg(employs.(\text{Emp where job = “clerk”}).sal}))
\]

The above query requires only inserting the condition \textit{job = “clerk”}. In SQL this is not so easy, because \textit{count} requires the \textit{group by} operator acting on the entire \textit{Emp} table, while \textit{avg} requires a similar grouping, but addressing only on a part of the \textit{Emp} table selected by the condition \textit{job = “clerk”}. This leads to a very awkward SQL query or some sequence of queries, with the help of materialized views. This can be considered another sign of the weakness of the \textit{group by} operator.

Below we give some quite easy (but a bit more sophisticated) examples in SBQL that for sure would require the \textit{group by} option in SQL, but cannot be so easily formulated - not excluding that due to the limitations of SQL they cannot be formulated at all (c.f. Fig.4.6, the object-oriented case).

E.4.42 For each location give the set of department names that are located at it, the average salary of bosses of these departments and the number of clerks that are (possibly) employed at these locations.

SBQL:

\[
(\text{distinct(Dept.loc as X). ((Dept where X in loc) group as Xdepts). (Xdepts.dname group as XdeptsNames,}
\text{avg(Xdepts.boss.Emp.sal) as XdeptsBossAvgSal,}
\text{count(Xdepts.employs.(Emp where job = “clerk”)) as XdeptsClerks# )}
\]

This query formulated in SQL requires two \textit{group by} clauses according to different criteria and each of the groupings should be differently named. This is impossible in SQL; hence the straightforward formulation of this query is in SQL impossible. Probably there is another formulation, for sure very complex and awkward.

E.4.43 For each salary range 0-99, 100-199, 200-299,…, etc. get the range, the number of employees getting the salary from the range and the average salary of bosses of
the employees getting the salary from the range. Formulate the result as report messages. Ranges with no employee belonging to should be omitted.

SBQL:

```
distinct( integerOf(Emp.sal/100) ) as R.
((100*R) as Rmin join (Rmin+99) as Rmax join
(Emp where sal ≥ Rmin and sal < Rmax) group as Remps).
(count(Remps) + “ employees earn between “ + Rmin +
“ and “ + Rmax + “; the average salary of their bosses: “ +
avg(distinct(Remps.worksIn.Dept.boss.Emp).sal))
```

The first line calculates the range discriminators $R$. The second line calculates the minimal and maximal salaries within a particular range. The third line calculates identifiers of employees that earn salary within the given range and forms from such employees a group named $Remps$. Three next lines form the output message for the given range using $+$ as a concatenation of strings.

Some professionals can consider as a positive property of the \textit{group by} operator some performance gain. Because the operator is closely related to its physical implementation, it can be tuned to be very efficient. This argument, however, works only for simple cases, when the grouping concerns stored tables. For tables that are calculated in a query the argument may be no more valid. Moreover, this argument reminds the old argument that programming should be done in assembler, because it is the most efficient and programs can be tuned by the programmers with no limits. The current point of view that governs the development of computer languages emphasizes human and economic aspects of programming processes, conceptual closure (orthogonality) of programming options and achieving proper quality of programs. Performance is very important, but it is the subject of internal optimizations rather than the conceptual construction of the language. In particular, query optimization methods developed for SBQL (and further methods) give some promises that the overall performance of SBQL queries that in SQL require grouping would be at least not worse. This of course requires testing and very detailed analysis of situations in query processing that are disadvantageous for performance.

As a side effect of our attempts to define the \textit{group by} operator for object-oriented store models we concluded that another grouping operator is necessary. This operator is denoted \textit{group as}; see examples E.4.37, E.4.42 and E.4.43. The operator groups a result returned by a query under a single name; then the group can be manipulated as a single element. The \textit{group as} operator is earlier known as the “nest” operator. Despite the origin, the \textit{group as} operator has semantically almost nothing in common with the \textit{group by} operator. The \textit{group as} operator is analogous to grouping some set of figures into a single figure, as for instance, in Power Point. The semantics of the \textit{group as} operator is very simple and it is explained in the section devoted to algebraic operators in SBQL.
5 SBQL for AS1 and AS2 Store Models

5.1 SBQL for AS1 Store Model

In the AS1 store model we have introduced a set of objects C representing classes, a relation CC representing inheritance among classes and a relation SC representing the membership of objects in classes. In this section we specify how these new features can be involved into the mechanism of query evaluation based on SBA. In fact, the corresponding extensions of semantics and implementation mechanisms are simple and natural. We have to consider possible corrections to the features of SBA that we have introduced to the model AS0, namely:

- The domain Result of results returned by queries;
- The construction of the ENVS stack;
- Rules for binding names on the ENVS stack;
- The construction of QRES stack;
- Function nested;
- Procedure eval for evaluation of queries.

Among six presented features only the last one is affected by the change of the store model from AS0 into AS1. The idea is that for a non-algebraic SBQL operator that acts on an object that is a member of some classes the ENVS stack is augmented not only by the internal features of this object, but also by the internal features of all the classes that the object belongs to. The internal features of an object and of its classes are calculated by the same function nested and pushed on ENVS in the proper order, which reflects a psychological or ergonomic view of the programmer: the object being processed is most close to his/her imagination, and then he/she considers its direct class, then its superclass, etc. In this way invariants of the object that are stored within its classes would become available for binding during processing of the object by a non-algebraic operator. The situation on ENVS during processing an object by a non-algebraic operator is presented in Fig.5.1.

For instance, consider the situation presented in Fig.2.8 and assume that we have a query:

\[(\text{Emp where } \text{name} = \text{“Poe”}) \cdot (\text{name, netSal, age})\]

In Fig.5.2 we present the situation on ENVS when the dot operator processes the Poe’s object. In the subquery after the dot name is bound on the top section of the stack (binders to internal properties of the Poe’s object), netSal is bound in the section below the top (binders to internal properties of the \text{EmpClass}), and age is bound in the third section from the top (binders to internal properties of the \text{PersonClass}).
Let us assume that a function \( \text{allClasses}(e) \) acts on any query result. For \( e \) being a single object identifier it returns sequence \( \{ c_1, c_2, ..., c_k \} \) of identifiers of the classes such that:

\[
\langle e, c_k \rangle \in \text{SC}, \langle c_k, c_{k-1} \rangle \in \text{CC}, ..., \langle c_3, c_2 \rangle \in \text{CC}, \langle c_2, c_1 \rangle \in \text{CC}
\]

In other words, the function \( \text{allClasses}(e) \) returns the identifiers of all the classes that the object identified by \( e \) is a member, in the order from the most general class \( c_1 \) till the most specific class \( c_k \). For \( e \) being \( \text{struct}\{ i_1, i_2, ..., i_t \} \) we assume that:

\[
\text{allClasses}(e) = \text{allClasses}(i_1) \circ \text{allClasses}(i_2) \circ ... \circ \text{allClasses}(i_t)
\]

where \( \circ \) is a concatenation of sequences. For all other forms of \( e \) the function \( \text{allClasses}(e) \) returns the empty result. So far we do not consider multiple inheritance and the situation when
classes identified by $e = \text{struct} \{ i_1, i_2, ..., i_t \}$ are in some conflict. We discuss the cases a bit later.

The corresponding part of the procedure `eval` that processes a non-algebraic operator can be described as follows.

```plaintext
procedure eval( q : query ) // change for the AS1 store model
begin
    ......
    case q is recognized as $q_1 \theta q_2$:
        // the query $q$ consists of a non-algebraic operator $\theta$ joining subqueries $q_1$ and $q_2$
        begin
            partialResults: bag of Result;
            partialResult, finalResult, $e$: Result;

            partialResults := bag{}; //empty bag
            eval( q_1 ); //Evaluation of the first query; the result is at the top of QRES
            for each $e$ in top( QRES ) do
                begin
                    for each $c$ in allClasses($e$) do
                        push( ENVS, nested($e$) ); //pushing internals of classes at ENVS
                    push( ENVS, nested($e$) ); //new section at ENVS
                    eval( q_2 ); //Evaluation of the second query; the result is at the top of QRES
                    partialResult := partialResultOf$_\theta$( $e$, top(QRES) );
                    partialResults := partialResults $\cup \cup \cup \{ partialResult \}$;
                        // new partialResult included to the partialResults bag
                    pop( QRES ); //removing the result($q_2$) from QRES
                    pop( ENVS ); //removing the top section with nested($e$) from ENVS
                    for each $c$ in allClasses($e$) do
                        pop( ENVS ); //removing class sections from ENVS
                end;
            finalResult := mergePartialResults$_\theta$( partialResults ); // forming the final result
            pop( QRES ); //removing the result($q_1$) from QRES
            push( QRES, finalResult);
                //the final result of the entire query is pushed at the top of QRES
        end;
    ......
end
```

New parts of the `eval` procedure are distinguished. These simple changes, plus the mechanism of invoking methods (see below), are the only amendments to the SBQL query engine. In the following we show that the AS1 model implies some problems with multiple inheritance and with collections, hence some next decisions and improvements can be necessary.

Our policy of pushing internals of classes on ENVS automatically supports the property known as **overriding**. For instance, if a method `income` is defined for both classes, `PersonClass` and `EmpClass`, then in the query `Emp where income > 2000` the name `income` is bound to the `income` method that is a property of the `EmpClass`; the `income` method within the `PersonClass` is invisible (is overridden). The policy in general supports the substitutability principle. In particular, if in a correct query having name `Person` one substitutes this name by `Emp`, the query is still correct; for instance, `Person where age > 20` and `Emp where age > 20`
are both correct queries. Substitutability, however, leads to some semantic problems with collections and with updating operations. We will return to this issue later.

5.1.1 Invoking Methods in AS1

Till now we have assumed that binding a name is simply a search within ENVS for a proper binder (binders) and then, return its value (values). In case of binding names of methods this action is extended by invoking the method. So far we are unable to present all the semantics related to invocation of methods; this will be done in the section devoted to imperative (procedural) abstractions of SBQL. Below we roughly explain what should happen when the name \( m \) of a method is to be bound:

- Before the binding of the method its parameters are evaluated and stored at QRES.
- The name \( m \) is bound on ENVS as previously.
- After the binding the method \( m \) is invoked. The invocation requires a new section that is pushed at the top of ENVS (so-called *activation record*).
- The new section contains binders to the local environment of the method \( m \), which consists of binders to the actual method’s parameters (taken from QRES) and binders to local method’s objects. Local objects and (sometimes) values of parameters are stored in a special (volatile) part of the object store. The section contains also so-called *return track*, i.e. information on the place of the program where the method was invoked.
- The program control goes to the method body.
- When the control leaves the method body, the mechanism performs the following actions:
  - If the method is terminated by a \textbf{“return }q\textbf{”} command, where \( q \) is a query, \( \text{result}(q) \) is pushed on QRES as usual.
  - The volatile part of the object store dedicated to this method is cleaned.
  - The return track is recorded, and then the new section is removed from ENVS.
  - The control goes to the method calling program according to the return track.

The action is a bit more complex if the class that the method \( m \) belongs to is subdivided into public and private parts; we return to this issue when we consider the AS3 store model. Note also that during executing of the method’s body a part of the ENVS stack should be invisible due to the lexical scope rules; this issue is also discussed later.

For instance, let us to consider a class \texttt{MyClass} that contains a method \texttt{MyMethod} having the following code:

```plaintext
method MyMethod( p1: T1, p2: T2): T {
    x1: T3; x2: T4;
    method body;
    return q;
}
```

\texttt{MyMethod} has two parameters: \( p_1 \) of type \( T_1 \) and \( p_2 \) of type \( T_2 \), and returns the result (determined by query \( q \)) of type \( T \). \texttt{MyMethod} declares two local objects \( x_1 \) of type \( T_3 \) and \( x_2 \) of type \( T_4 \). Let \texttt{MyMethod} be invoked in a query:

\[ q_0 \text{ where } \text{MyMethod}(q_1, q_2) = 1000 \]
where \( q_0 \) is a query returning identifiers of objects being members of the class \( MyClass \), \( q_1 \) and \( q_2 \) are actual parameters of \( MyMethod \), which is called in the context of the non-algebraic operator \( where \) (the same reasoning is for other non-algebraic operators and for any other query involving \( MyMethod \) after such an operator). Let \( i_{MC} \) denotes the identifier of the \( MyClass \) object and let \( result(q_0) = \text{bag}\{r_1, r_2, \ldots\} \). Fig.5.3 shows some states of \( ENVS \) (above the \( time \) line) and \( QRES \) (below the \( time \) line).

The above picture will be discussed in the section devoted to imperative abstractions, where we will consider methods of parameter passing and general scope rules.

### 5.1.2 Multiple Inheritance

The definition of the AS1 store model allows for multiple inheritance among classes and multiple memberships of objects in classes. The multiple memberships can be reduced to multiple inheritance, hence we do not consider this case separately. If we allow that a class can inherits from two or more super-classes, in general it is possible that there will be a conflict. Let class \( C \) inherits from \( A \) and \( B \). If \( A \) contains a method named \( m \) and \( B \) contain a method named \( m \), then the class \( C \) can inherit only one of the methods; the other one will be invisible. In the AS1 model there is no cure for such a situation if we assume the open-close principle; i.e. when classes \( A \) and \( B \) are developed independently, e.g. by different external companies and their source codes are unavailable. In this case the class \( C \), which inherits both from \( A \) and \( B \), but one of the methods cannot be inherited, violates the substitutability principle. For instance, let \( C \) inherits the method \( m \) from \( A \), but not inherits the method \( m \) from \( B \). In such a case objects of the class \( C \) cannot be considered as objects of the class \( B \). Some proposals allow in such a case for changing (virtually) during inheritance the name of
the method \( m \) for the class \( C \); for instance, the method \( m \) that is not inherited from \( B \) during inheritance receives a virtual name \( mB \). This option solves the problem only partially, when a given object is considered as a member of \( C \). When an object of the class \( C \) is considered as an object of the class \( B \) (due to substitutability) it still requires the method \( m \), which will be taken from \( A \) rather than from \( B \).

Despite this anomaly, multiple inheritance is a very useful option for conceptual modeling; hence it is not a good strategy to forbid it. In particular, multiple inheritance makes it possible to involve abstract classes, i.e. classes having no members, including so-called \textit{mixin classes}, i.e. classes storing invariants of many different classes. The model with multiple inheritance is introduced in many practical proposals, including programming languages such as C++, the CORBA standard, the ODMG standard, etc. Smalltalk and Java, however, do not introduce multiple inheritance, what can be considered disadvantageous for conceptual modeling. In programming languages such as C++ multiple inheritance leads also to some problems with physical representation of objects.

In our case the multiple inheritance can be involved into the definition of the function \textit{allClasses}, which returns the sequence of class identifiers for a given object (objects), in the reverse order. In case of multiple inheritance there are many such possible orders and the algorithm for calculating the function must choose one of them. It is not important which of the orders is to be taken, because in the case of conflicts between names of methods no good order exists.

A problem with multiple inheritance is the result of mixing in one environment properties coming from different independently developed classes. Such a mix must sometimes lead to naming conflicts. The problem has a radical solution in the AS2 store model, which allows one to substitute multiple inheritance by dynamic object roles and dynamic inheritance.

A problem similar to multiple inheritance appears when the function \textit{allClasses(e)} acts on \( e = \text{struct} \{ i_1, i_2, ..., i_t \} \), where \( i_1, i_2, ..., i_t \) are identifiers of objects being perhaps members of different classes. In this case the conflict between names of methods occurring in these classes is also possible. This time, however, the conflict can be explicitly removed by the programmer by using proper SBQL options, for instance, by auxiliary naming.

### 5.1.3 Collections in the AS1 Store Model

The biggest problem with the AS1 model concerns collections. The problem is recognized, in particular, in the ODMG standard. There is a conflict between substitutability and collections. According to substitutability, each \textit{Emp} object is also a \textit{Person} object. However, in the AS1 model each object has one name hence it belongs to one collection. For instance, if some non-algebraic operator processes \textit{Person} objects, one can expect that it automatically will take also \textit{Emp} objects residing in the same environment (e.g. a database). However, the standard binding mechanism disallows for that. For example, (c.f. Fig.2.8) the query

\[
\text{Person where age < 20}
\]

will take into account the Doe’s object only. Poe and Kim will not be processed, because the name of corresponding objects is \textit{Emp} rather than \textit{Person}.

One can consider in such a case that the query can be formulated as follow:

\[
\text{bag(Person, Emp) where age < 20}
\]

Such a solution is not only awkward. It still does not solve the problem due to the open-close principle. Assume that one writes a \textit{PersonCollectionClass} for the \textit{PersonClass},
implementing all the required methods. Then, much later someone defines a specialized class `EmpClass` class that inherits from `PersonClass`. After that, some amendments must be introduced to the `PersonCollectionClass` and the class must be recompiled. This can be impossible if the source code of the `PersonCollectionClass` is unavailable. Anyway, the open-close principle is violated.

There are several solutions of this problem, but all of them require some additional assumptions concerning the AS1 model and some changes to the binding mechanism. In particular, we can assume that each object has as many external names as classes it belongs to. For instance, (c.f. Fig.2.8), we can assume that the Poe’s and Kim’s objects have additionally the external name `Person`. The binding mechanism is modified in such a way that whenever a binder `Emp( oid )` is put on ENVS, the binder `Person( oid )` is put on ENVS too. For instance, the bottom section of ENVS presented at Fig.4.9 will have the binders:

```
Person(i1), Person(i2), Person(i3), ... , Emp(i5), Emp(i9), ...
```

The strong typing mechanism should assure in such cases that specific properties of `Emp` cannot be used when the object is regarded as `Person`.

Alternatively, we can assume that each concrete class contains a name of its members as an invariant. This method does not require assigning many names for an object. For instance, the `PersonClass` requires that all the members have the name `Person`, and the `EmpClass` requires that all the members have the name `Emp`. The binding mechanism is modified in such a way that, for instance, when the name `Person` is bound first the class containing such name as an invariant is identified, `PersonClass` in this case. Then, according to the CC relation, all classes that inherit from the `PersonClass` are identified, in particular, the `EmpClass`. Then, the `bind` function is extended to `bindAll` function that takes all invariant names of objects from all the subclasses of the given class. For instance, it holds:

```
bindAll( Person ) = bind( Person ) ∪ bind( Emp )

bindAll( Emp ) = bind( Emp )
```

In further examples concerning SBQL for the AS1 model we assume this method. Classes with invariant names of their member objects are implicitly assumed in UML class diagrams and in the ODMG standard.

Such a binding strategy solves the problem, but the requirement that each concrete class must possess an invariant name for all its members may introduce limitations for some applications. As some cure for this inconvenience, we can assume the possibility that a specialized class may “override” this invariant, in particular, drops it. For instance, an `EmpClass` having `Emp` as the invariant object name can be specialized by `AnyEmpClass`, which inherits everything from `EmpClass` but the invariant object name. Alternatively, we can assume that the name of an object is determined by its class, but the programmer can change it for a concrete object (after such a change the object is not seen as a member of an extent of the class, although inherits from it all the properties). Another solution is that an object name is determined by its class, but the programmer can assign to the object an arbitrary alias. Yet another solution is that an invariant name determined by a class concerns only persistent (database) objects and do not concern volatile objects being local to a user session or belonging to local environments of procedures or methods. There are perhaps other solutions. Without experiments made for real applications is hard to say which solution is the best for further programmers.

In AS1 there is also a problem with typing of pointer objects. In the tradition of programming languages, a pointer object is specified by the type of an object that it points too. The tradition
of databases is that a pointer link is specified by the name of an object that the pointer link leads to. Such an assumption is necessary if one wants to specify precisely a database schema, which (as we have discussed previously) is an inherent pragmatic part of any query language. The assumption is explicitly taken by the ODMG standard and implicitly by a lot of other proposals aiming specification of query languages. Trying to make a coincidence of these two ideas we can say that a pointer link should be specified both by the type of an object it leads to and by the name of the object. Assuming that an object name is an invariant for a class we can reconcile both ideas with no such pain: pointer objects will be specified by the name of the class of an object that the pointer leads too, and the class will contain both the type of the object and its invariant name. More detailed discussion on typing issues for SBQL we present later.

The AS1 has also other disadvantages, for instance, lack of multi-aspect inheritance, lack of repeating inheritance, etc. As in the case of the problem with multiple inheritance, the radical solution to all the problems with the AS1 store model is the AS2 store model.

5.1.4 Examples of SBQL Queries for the AS1 Store Model

Fig.5.4 presents an object-oriented database schema as an UML-like class diagram. Note that cardinalities are assigned not only to relationships, but to any object declaration, to any attribute, etc. Lack of cardinalities means the default cardinality [1..1]. Relationships such as worksIn/employs should be understood as declarations of pointer objects. Address is a complex attribute (which cannot be directly expressed in UML). Person, Emp and Dept are invariant names of the member objects for the corresponding classes. Binding of name from a super-class means implies binding all names from its sub-classes; for instance, binding of Person implies additional binding of Emp. The method budget returns the annual budget of a department.

![Fig.5.4. Database schema according to the AS1 store model](image-url)
E.5.1 Get names of departments and the average age of their employees (inheritance of the method age).

SBQL: \( \text{Dept}. \text{struct}(\text{dname, avg(employs.Emp.age())}) \)

SBQL: \( \text{Dept}. (\text{dname, avg(employs.Emp.age())}) \)

SBQL: \( \text{Dept}. (\text{dname, avg(employs.Emp.age()) as avgAge}) \)

In general, methods having no parameters can be written without parentheses, for instance, \( \text{age} \) is an equivalent to \( \text{age}() \). The last query returns a bag of structures \( \text{struct}\{i_{\text{Dept}}, \text{avgAge(some_real)}\} \) consisting of the identifier of a department \( i_{\text{Dept}} \) and a binder with name \( \text{avgAge} \) and some real value.

E.5.2 Get name, the net salary and the boss name for programmers working in departments located in “Paris” (inheritance of name).

SBQL: \( (\text{Dept where } \exists \text{ loc as } x (x = \text{“Paris”}))) \)
join \( (\text{employs.Emp where } \text{“programmer” in job}). \)
\( (\text{name as name, netSal as netSal, (boss.Emp.name) as boss}) \)

The query returns a bag of structures having three binders: \( \text{struct}\{ \text{name(i_{name1}), netSal(some_real), boss(i_{name2})}\} \).

E.5.3 Get employees that for sure live in the cities where their departments are located (inheritance of Address).

SBQL: \( \text{Emp where } \exists \text{ Address as } a (\exists (\text{worksIn.Dept.loc as } l (a.\text{city} = l))) \)

E.5.4 For each employee get the name and the percent of the annual budget of his/her department that is consumed by his/her salary.

SBQL: \( \text{Emp}. (\text{name as } n, (((\text{if exists(sal) then sal else 0}) as } s). \)
\( ((s \times 12 \times 100)/((\text{worksIn.Dept.budget})) \text{ as percentOf Budget}) \)

E.5.5 Imperative construct: for each person having no salary give the minimal salary in his/her department.

SBQL: \( \text{for each (Emp where not exists(sal)) as } e \text{ do } \)
\( e.\text{changeSal( min(e.works_in.Dept.employs.Emp.sal) );} \)

In this example we have assumed that the method \text{changeSal} \) inserts a subobject \text{sal} \) with a proper value into the given \text{Emp} \) object. We also assume that for each department there is at least one employee having the salary. Note that the program is not optimal - it counts the minimal salary for each employee having no salary. More optimal program should calculate the minimal salaries for proper departments, then get it to proper employees, for instance:

SBQL: \( \text{for each (Dept where } \exists \text{ employs.Emp (not exists(sal)) as } d \text{ join } \)
\( \text{min(d.employs.Emp.sal) as minSal do } \)
\( \text{for each (d.Employ.Emp where not exists(sal)) as } e \text{ do } \)
\( e.\text{changeSal( minSal );} \);
This manual optimization can be substituted by an automatic optimization of the first SBQL program, but developing a proper algorithm could be a challenge.

5.2 SBQL for the AS2 Store Model

To the best of our knowledge, among all the proposed query languages only SBQL addresses the model with dynamic object roles and dynamic inheritance. This feature was implemented (several times) in our prototypes, for instance, in for the needs of PhD-s by A.Jodlowski [Jodl03a] and R.Adamus [Adam06]. Other proposals do not even noticed that such a model can be the subject of query languages. However, many features of currently considered artifacts, such as executable UML in the MDA architecture, call for such a capability. Similar features sooner or later will be necessary for Web-oriented ontology descriptions. In our opinion, a sufficiently general query language having such a feature cannot be developed on the ground of other theoretical approaches to query languages. Changes introduced by AS2 in comparison to the AS0 and AS1 models concerns essentially three points, which are rather conceptually obvious and easy in implementation.

1. An initial database section on ENVS should contain binders to all the roles.
2. When a non-algebraic operator pushes on ENVS a section for a role, it automatically pushes on it sections of all its super-roles, in proper order.
3. Some new operators in SBQL are necessary to serve roles.

Fig.5.5 presents the beginning state of ENVS for the database shown in Fig.2.10 and Fig.2.11.

In more formal terms, if the database is described by the AS2 store model \( <S, C, R, CC, SC, SS> \), then the database section is filled in by binders all binders \( n(i) \), where \( i \in R \) and \( n \) is the name of the object (perhaps a role) \( <i, n, x> \in S \).

Concerning the second change, it implies pushing sections of super-roles when a role section is pushed on ENVS. Consider Fig.2.10, Fig.2.11 and a query having the form:

\[ Emp \text{ where } ... n \text{ ...} \]

where \( n \) is any name occurring in the subquery after \textbf{where}. We assume that the \textbf{where} operator currently processes the \textit{Emp} role identified by \( i_{16} \). The state of ENVS is presented in Fig.5.6.
Fig. 5.6. State of ENVS during processing a role *Emp* by a non-algebraic operator

Fig. 5.7 presents a more general case when an object contains 5 roles connected to some classes. Classes are connected by inheritance links.
Assume that currently the role R5 is processed by a non-algebraic operator. The situation of ENVS is presented in Fig.5.8.

All the 9 sections, in the order determined by Fig. 5.8, are pushed by a single non-algebraic operator. All of them are removed when the operator finishes its job. Developing a general case, on the ground of the given relations CC, SC, SS, is a rather easy implementation issue.

Note that some sections may appear more than once. This is not a problem, because only the section closest to the top is taken into account. Optimization of this case, by removing unused sections, is possible, but one should not expect a big gain from it.

Note that (unlike AS1) the AS2 model does not require that object names are to be invariant of classes. At the same time it allows to create and process heterogeneous collections. Obviously, through its roles an object can simultaneously belong to many collections. Still, however, there is the same problem as in AS1 concerning typing pointers leading to an object with a given name.

5.2.1 Special SBQL Operators for the AS2 Model

According to the conceptual closure principle, roles, as a new feature of store model, require some new features in a corresponding query/programming language. In SBQL we propose the minimal and sufficient set of such features. Concerning the pure querying, we see only the need for two new operators:

- A dynamic cast operator which converts a reference to a role into a reference (references) of another role (roles) of the same object. The operator will be written before a query returning such a reference (references) as
(role_name) query

- If the object has no a role role_name, the operator returns an empty bag. The operator is defined as macroscopic: it may act on a bag of references to roles and returns a bag being the union of all the references that are returned for individual arguments. The operator works in any direction: it may be used to identify a super-role, a sub-role or a neighbor role.

- Testing if an object has a given role. The syntax is the following:

  \[ \text{query has role role_name} \]

  where query returns a single reference to a role. The operator returns true if the role identified by query has a sub-role named role_name, and false otherwise. If query returns an empty bag or it returns more than one role, this is probably an error thus the system should throw an exception. Alternatively, we can assume existential quantification, that is, the following equivalence:

  \[ \text{query has role role_name} \quad \text{is equivalent to} \quad \exists x \,(x \text{ has role role_name}) \]

  Assuming dynamic programming with reflection other operators can be necessary, such as returning names of all roles of the given object.

There is very low demand for new data manipulation operators specific for dynamic object roles. Actually, only one such operator is indeed necessary, that is, insert an object as a sub-role of the given role. The reverse delete operator is a bit extended: removing a role implies removing all its sub-roles. In particular, removing a main role is equivalent to removing the corresponding object. Roles, however, may present some problem for virtual updateable database views. Concerning this topic, no research, implementation or proposal is done.
5.2.2 Examples of SBQL Queries for the AS2 Model

In Fig.5.9 dynamic inheritance is denoted by an arrow with a black and white diamond end. Cardinalities express the facts that there can be any number of persons, each person can be an employee at most once and a student zero or more times. Each Student role is connected to exactly one school. Each Emp role is connected to exactly one Dept. An Emp role can have a Manager sub-role; it has no attributes.

### E.5.6 Get employees older than 60 who live in Warsaw (dynamic inheritance of the attribute Address and the method age).

**SBQL:**

```
Emp where age > 60 and \( \exists \) Address (city = “Warsaw”)```

### E.5.7 Get names and net salaries of managers managing departments located in “Cracow” (dynamic inheritance of the attribute name from the Person class, static inheritance of the method netSal from the Emp class and navigation to Dept).

**SBQL:**

```
(Manager where \( \exists \) ((manages.Dept.loc as l) (l = “Cracow”)) . (name, netSal)```

### E.5.8 Get names of persons who are at the same time employees and students (dynamic casts or the has role operator).

**SBQL:**

```
(Person)((Emp)Student)
SBQL: ((Person as p) where p has role Emp and p has role Student).p.name```
E.5.9 Get name, faculty and school name for each person studying at two or more faculties.

SBQL: 

\[
(((\text{Person as } p) \text{ join } (((\text{Student as } s) \text{ group as } s))) \text{ where } \text{count}(s) \geq 2). \\
(p.name, s.(\text{faculty}, (\text{studiesAt.School.name})))
\]

E.5.10 For each person get name and the sum of all the incomings (salary and scholarships).

SBQL: 

\[
(\text{Person as } p). (p.name, \text{sum}(\text{bag}(0, ((\text{Student as } s).\text{scholarship}, ((\text{Emp as } e).\text{sal})))))
\]

Note that the query takes automatically into account that some Person has no incomings. It correctly addresses the fact that he/she can have many Student roles, thus many scholarships. It also works for the case when his/her role Emp role has no sal attribute. Compare an “equivalent” SQL query to realize that it is much more complex and illegible than the SBQL query. We assume the relational schema that converts a part of the above dynamic roles schema: Dept(id, name), Person(id, name), Emp(id, Dept_id, Person_id, sal), Student(id, University_id, Person_id, scholarship)\(^8\).

SQL: 

\[
\begin{align*}
\text{SELECT name, SUM(income)} \\
\text{FROM Person } pe, \\
(\text{SELECT Person_id pid, scholarship AS income FROM Student}) \text{ AS v} \\
\text{WHERE v.pid = pe.id} \\
\text{GROUP BY v.pid, pe.name;}
\end{align*}
\]

E.5.11 Get students who live in the same city as the city of their school.

SBQL: 

\[
\text{Student where } \exists \text{Address (city = (studiesAt.School.city))}
\]

5.3 SBQL order by Operator and Range Queries

A sorting operator order by is introduced in SQL and is proposed in other query languages, in particular in ODMG OQL. The operator makes it possible to determine sorting of a query result according to some chosen keys. Sorting is a very important operation in query languages for the following reasons:

- Output of query results should be sorted for increasing legibility and enabling quick search in long reports by humans.

- Sorting increases the power of the query language. Some important queries cannot be formulated without sorting. Such queries are known as range queries. For instance, a range query is “Get 50 best-paid employees”.

In the relational model sorting was considered auxiliary, implementation feature, which can be used only for forming final output from a query. The sorting cannot be covered by the relational model. Relations are sets and the order of tuples in a relation is inessential. Hence

\(^8\) The example is due to Krzysztof Stencel.
the sorting operator and operations relying on sorted collections are not expressible in the relational model theory. They are also not expressible in other theories developed for the relational model, such as relational calculus and formal logic. Fortunately, majority of implementations of SQL do not consider theories as a serious obstacle in developing programmer’s interfaces. Some SQL implementations provide also range queries allowing to retrieve $i$-th element from a sorted relational table (or to retrieve elements from $i$-th to $j$-th).

In object-oriented databases sorting operator is provided as well. In ODMG OQL the sorting operator is introduced exactly as in SQL, including range queries. Object-oriented databases usually introduce sequences as a kind of stored collections, hence sorting operator and range queries become necessary. Similarly, XML assumes the order of its parts; hence languages such as XQuery can be used to formulate range queries.

However, some aspects of the sorting operator and range queries are not considered systematically in the literature and in proposed languages. Some range queries are still impossible to formulate. For instance, consider a query “Get employees in the alphabetic order according to last names, together with the rank number of their salaries (the lowest salary obtains the rank number 1, the highest salary obtains the rank number $\text{count}(\text{Employees})$)”. Such a query cannot be formulated in SQL, OQL and XQuery. The reason is that in these languages a rank number can be used in queries as input, but not as output. SBQL is the first query language that allows for both cases.

In this chapter we present thoroughly major issues related to the sorting operator and range queries in the object query language SBQL.

### 5.3.1 Sorting Operator in SBQL

In SBQL the sorting operator belongs to non-algebraic operators. We assume the following syntax:

$$query ::= \text{query order by query}$$

Semantics of the query $q_1 \text{ order by } q_2$ is as follows. As usually, a collection being sorted is determined by $q_1$ and a sorting key by $q_2$.

The precise semantics is explained in the following steps:

First $q_1 \text{ join } q_2$ is evaluated (see the semantics of the non-algebraic join operator). Let $q_1$ returns bag\{ $r_1, r_2, \ldots$ \} or sequence\{ $r_1, r_2, \ldots$ \}, a sequence is converted to bag. In the result of the join we obtain a bag of structures

$$\text{bag}\{ \text{struct}\{ r_1, v_{11}, v_{21}, \ldots, v_{k1} \}, \text{struct}\{ r_2, v_{12}, v_{22}, \ldots, v_{k2} \}, \ldots \}$$

where $\text{struct}\{ v_{ji}, v_{2j}, \ldots, v_{kj} \}$ is a structure returned by $q_2$ for $r_i$.

If some $v_{ji}$ is a reference, a dereference operator is performed which changes the reference to a value. If a dereference concerns a reference to a complex object then it is treated as a failure. This situation can be considered a typing error discovered during static type checking. If the dereference is impossible or the value after the dereference does not belong to the type with a natural linear ordering, then such a situation is considered a failure too. For instance, this is the case of multimedia data types. This situation can also be considered as typing error discovered during static analysis of a query.

The bag resulted from $q_1 \text{ order by } q_2$ is sorted according to $v_{ji}$, then for identical $v_{ji}$ is sorted according $v_{2j}$, etc. till performing the sorting for the last sorting key $v_{ki}$.

After sorting we obtain a sequence
sequence{ struct{ s1, vs11, vs12, ..., vs1k1 }, struct{ s2, vs21, vs22, ..., vs2k2 }, .... } which differs from the previous bag only by the order of structures.

The final result of \( q_1 \) order by \( q_2 \)

sequence{ s1, s2, .... } is obtained from the previous sequence by removing results returned by \( q_2 \) (equivalently, projection of the sequence to the results returned by \( q_1 \)).

Examples (see Fig.5.4)

| E.5.12 | Get references to employees sorted according to names |
| SBQL:  | Emp order by name |
| Note that the query is formulated with the use of inheritance. |

| E.5.13 | Get references to employees sorted according to their ages and then according to names |
| SBQL:  | Emp order by (age, name) |
| Note the use of the inherited method. |

| E.5.14 | Get departments sorted according to the number of employees and then according to boss names; return names of departments, and their locations sorted alphabetically. |
| SBQL:  | (Dept order by (count(employs), (boss.Emp.name))). (dname, ((loc as x) order by x),x) group as locations) |
| We use an auxiliary name x as an ordering key. Then, to remove x from the result we make projection on x. All locations of a department are grouped under a single name locations. Note that in the result bag SBQL returns references rather than values. |

| E.5.15 | As above, but return only departments with the budget greater than 1000000. The output should contain the number of employees and name of a boss. |
| SBQL:  | (((Dept where budget > 1000000) join count(employs) as c join (boss.Emp.name) as b) order by (c, b)). (dname, c, b, ((loc as x) order by x),x) group as locations)) |
| We assume that all algebraic and non-algebraic operators preserve the order of elements in a processed sequence. This concerns such operators as \textbf{where}, \textbf{dot}, \textbf{as}, \textbf{for each}, etc. For some operators, e.g. for join, this is not obvious. In general, for each SBQL operator it should be determined how it behaves for combination of kinds of collections being its arguments. |
| Sometimes sorting of strings require regarding/disregarding small and capital letters. This effect can be achieved through typical operations on strings. |

| E.5.16 | Get references to employees sorted according to names; disregard lower and upper letter cases |
| SBQL:  | Emp order by toUpper(name) |
5.3.2 Empty and Multi-Valued Keys

The presented semantics of the \texttt{order by} operator has several peculiarities which should be well understood. The first of them concerns the situation when the sorting attribute is optional, for instance \texttt{sal} in Fig.5.4. Consider the query

\begin{itemize}
\item \textbf{E.5.17} Get references to employees sorted according to salaries
\end{itemize}

\texttt{SBQL: Emp order by sal}

According to the semantics of the \texttt{join} operator on which the \texttt{order by} operator is based on, the query omits all employees for which the subobject \texttt{sal} does not exist. Hence the sorted result will contain only references to \texttt{Emp} objects having \texttt{sal} attribute. If the programmer wants to take into account all \texttt{Emp} objects, he/she can use, in particular, one of the methods that are presented in the chapter devoted to storing and processing irregular data. For instance, assuming that for employees having no salary the sorting key is 0, the above query can be formulated as follows:

\begin{itemize}
\item \textbf{E.5.18} Get references to all employees sorted according to salaries; assume \texttt{sal} = 0 for employees having no salary
\end{itemize}

\texttt{SBQL: Emp order by max(bag(0, sal))}

\texttt{SBQL: Emp order by if exists(sal) then sal else 0}

A similar situation arises with multi-valued keys. Consider the query

\begin{itemize}
\item \textbf{E.5.19} Get references to employees sorted according to jobs
\end{itemize}

\texttt{SBQL: Emp order by job}

Attribute \texttt{job} is multi-valued, hence according to the join operator each reference to an employee object having \texttt{n} jobs, \texttt{n} > 1, will be repeated \texttt{n} times in the final result. Such interpretation of the sorting operator seems to be the most logical and reasonable, however, some programmers could be surprised with such a result, hence they should be aware of the semantics. Of course, there are many ways to avoid repetitions of references. For instance, the programmer can write a query in such a way that for sorting only the first job is considered (providing jobs are typed as sequences):

\begin{itemize}
\item \textbf{E.5.20} Get references to employees sorted according to jobs, return employee name and jobs (no repetition of \texttt{Emp} references).
\end{itemize}

\texttt{SBQL: (Emp order by job[1])(name, job group as jobs)}

If jobs are not typed as sequences, depending on the type constraint \texttt{job[1]} will return a type error or a random job. It is also possible to construct a query that takes some additional sorting logic. Assume there is a collection of objects \texttt{JobRange(j: string, r: integer)} that assign to jobs range number (1 is the range of the most important job, etc.). Now we can ask the following query:

\begin{itemize}
\item \textbf{E.5.21} Sort employees according to their most important jobs; return employee name and jobs sorted according to their range (no repetition of \texttt{Emp} references).
\end{itemize}

\texttt{SBQL: (Emp order by min((JobRange where j in job).r). (name, ((job as k order by ((JobRange where j = k).r).k) group as jobs})

The examples show that there are a lot of semantic peculiarities with the sorting operator. However, the power of SBQL gives the hope that almost all of them can be efficiently formulated in the query language.
5.3.3 Sorting in Ascending and Descending Order

In SQL and OQL there are special keywords asc and desc placed after a sorting key; asc is optional. Mathematically, asc is an identity function, while desc is a generic function that returns the reverse of an argument (assuming the linear ordering of the sorting key domain). The function desc for a number X is simply \(-X\). Alternatively, assuming that a sorting key should be always a positive number, we can assume that desc(X) = MaxPositiveNumber – X. For strings the function desc can be defined as a conversion of the string characters where each character with ASCII code X is substituted by a character with ASCII code 256 –X; similarly for other coding standards. The function desc should be defined for each domain that values can be sorting keys. It is a secondary issue if such a function should be written in a typical syntax desc(X) or in the syntax assumed by SQL and OQL: X desc.

E.5.22 Get references to employees sorted according to age, descending, and then according names, ascending.

SBQL: (Emp order by (desc(age), asc(name)))

5.3.4 Alphabetic Order in Native Languages

In early relational systems the SQL order by operator caused a lot of controversy concerning sorting strings in native languages different than English (especially in German and French). The SQL developers assumed the ASCII coding for English characters, which exactly corresponds to the alphabetic order of strings in English. However, the rules of alphabetic order for other languages (German, French, Polish, etc.) are different. Moreover, alphabets of these languages contain characters absent in the English alphabet. Sometimes sorting rules have anomalies, for instance in German double s is equivalent to the character \(\beta\). Changing the rules of ordering by SQL was totally unacceptable for many users (and office rules), who wanted to follow several hundred years of tradition concerning various kinds of alphabetically ordered lists, reports, dictionaries, etc. In effect, vendors of relational systems were forced to change the rules of sorting assumed by the SQL order by operator. The solution gives the decision in hands of the database administrator, who has proper options to determine the order of strings for a particular application according to the given native language. This method has solved majority of sorting problems, but not all of them. For instance, one application can require (in the same run) sorting according to English, according to German and according to Polish. Only the programmer of a database application (not the database administrator) is able to determine which kind of ordering is required in a particular place of the program.

There is a simple solution of the problem, which consists in implementing a family of functions with string arguments that return the ordering key corresponding to a particular native language. Such a function takes a parameter being a string in a native language and returns a string of numbers. For each character in the input string the function returns a number being its position in the alphabetic rank. For instance, for Polish the alphabet the function mapping can be as follows: a→1, á→2, b→3, c→4, ć→5, d→6, e→7, ę→8, f→9, g→10, h→11, ... This reminds the ASCII coding, but the system can be different than ASCII due to some exceptional cases (e.g. equivalence of –ss- and –\(\beta\)- in German). Because such functions are an internal sorting feature, correspondence to any standard is less essential.

An additional parameter of such a function can determine an encoding standard of an input string, for instance, ISO, UTF-8, etc.
E.5.23 Get references to employees sorted according to department names in English (ISO), and then, according to employee names in Polish (UTF-8).

SBQL: \((\text{Emp \, order \, by \, (English(ISO, worksIn.Dept.dname), Polish(UTF-8, name))})\)

5.3.5 Range Queries

In general, range queries are properties of sequences (as collections) rather than directly properties of ordering query operator. Because the \textit{order by} operator always returns a sequence, range queries are naturally associated with this operator. In the simplest variant a range query means the possibility to choice an \textit{i}-th element of a sequence, where \textit{i} is a parameter that can be determined by an expression (a query). The typical syntax is as follows:

\[
\text{query} \triangleq \text{query} [\text{query}]
\]

The query \(q_1[q_2]\) returns \(i\)-th element of \(q_1\), where \(i\) is determined by \(q_2\). In a more general variant, \(q_2\) can return a bag of numbers. In this case \(q_1[q_2]\) returns all elements that are pointed by \(q_2\). We can also assume cardinalities written in the form \(i..j\) or \(i..*\), which return a bag of numbers starting from \(i\) and ending by \(j\), or starting by \(i\) and ending by infinity. This convention gives us a lot of possibility to write range queries, for instance:

E.5.24 Get references to 50 best-paid employees.

SBQL: \((\text{Emp \, order \, by \, desc(sal))}[1..50]\)

E.5.25 Get the median salary of employees

SBQL: \( ((\text{Emp \, order \, by \, sal} ))((\text{integer})(\text{count(Emp)}/2)))\).sal

Note that in the above examples we assume that counting of sequence elements starts from 1. The languages C, C++, Java, C# and standards CORBA and ODMG assumed that the counting starts from 0. This solution was reasonable in assembler and C/C++ due to accordance of indices of arrays with pointer arithmetic. However, currently developed languages are very far from assemblers, hence such a convention is a kind of unnecessary atavism what is illogical and error prone. In SBQL we have rejected it.

E.5.26 Get references to 50 best-paid employees and 50 worst-paid employees.

SBQL: \((\text{Emp \, order \, by \, desc(sal)}))[\text{bag}(1..50, (\text{count(Emp)} – 49)..<\text{count(Emp)})]]\)

If we would use the convention that ordering of sequences starts form 0, the above query must be reformulated to a less legible form:

E.5.27 Get references to 50 best-paid employees and 50 worst-paid employees.

\((\text{Emp \, order \, by \, desc(sal)}))[\text{bag}(0..49,(\text{count(Emp)}-50)<..(\text{count(Emp)}-1))]\)

Queries written in this syntax are able to return elements of a sequence if their order numbers are known, but are unable to return order numbers for sequence elements that are selected in another way. For instance the query “Which rank number takes Brown in the ranking concerning salary?” is to be formulated as a program (a sequence of instructions) rather than a single query. For this reason in the Loqis and ODRA system we have introduced a special operator \texttt{range as}. The syntax is similar to \texttt{as} and \texttt{group as} operators:

\[
\text{query} \triangleq \text{query} \, \text{range as name}
\]
Semantics of the query \( q \text{ range as } n \) is as follows. Let \( q \) returns \( \text{sequence}\{ r_1, r_2, r_3, \ldots \} \). Then, the query \( q \text{ range as } n \) returns

\[
\text{sequence}\{ \text{struct}\{ r_1, n(1) \}, \text{struct}\{ r_2, n(2) \}, \text{struct}\{ r_3, n(3) \}, \ldots \}
\]

Each element \( r_i \) returned by \( q \) is associated with the binder \( n(i) \), where \( n \) is a name determined by the programmer, \( i \) is a rank number of this element in the sequence, starting from 1. This makes it possible to formulate any conditions based on this rank number, as well as returning it as an output of a query.

<table>
<thead>
<tr>
<th>E.5.28</th>
<th>Which rank number takes Brown in the ranking concerning salary?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>(((\text{Emp order by desc}(\text{sal})) \text{ range as } s) \text{ where } \text{name} = \text{``Brown''}.s)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E.5.29</th>
<th>Get a report sorted by department names and returning the names and the rank of the department concerning the number of employees, descending.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>(((\text{Dept order by desc}(\text{count}(\text{employs}))) \text{ range as } c) \text{ order by } \text{dname} \text{``Brown''}.(\text{dname}, c))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E.5.30</th>
<th>Return employees having the rank in the salary category (descending) at least on 10 higher from the rank in the age category (descending):</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>(((\text{Emp order by -sal}) \text{ as } e \text{ range as } i) \text{ order by -e.age} \text{ range as } j) \text{ where } i &gt;= j+10).e)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E.5.31</th>
<th>Return the average salary of employees, disregarding 25% of the worst paid and 25% of the best paid ones.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>(\text{avg}(((\text{Emp order by sal}) \text{ range as } r) \text{ where } r &gt; (\text{integer})(0.25 * \text{count}(\text{Emp}))) \text{ and } r &lt; (\text{integer})(0.75 * \text{count}(\text{Emp}))).\text{sal})</td>
</tr>
</tbody>
</table>

The examples have shown that the operator such as \text{range as} can be a very useful addition to any query language.
6 Imperative Constructs in SBQL

In SBA we reconstruct query languages’ concepts from the point of view of programming languages (PLs). The approach is motivated by our belief that there is no definite border line between querying and programming; thus there should be a universal theory that uniformly covers both aspects. The design of modern and universal database PLs having querying capabilities requires methods and principles that are already acknowledged by the common practice of developing compilers and interpreters. Practically useful PLs deal with object-oriented notions (classes, methods, inheritance, etc.), procedures and functional procedures (including recursive ones), parameter passing methods, various control statements, binding strategies, scoping rules, modularization, strong typing, etc. They should follow software engineering principles such as orthogonality, modularity, minimality, universality, genericity, typing safety, and clean, precise semantics. SBA is an alternative to theoretical concepts emerging on the object-oriented wave, such as nested relational algebras, object algebras, object calculi, F-logic, comprehensions, structural recursion, monoid calculus, functional approaches, etc. Careful analysis of them and our implementation experience convinced us that all of them are limited and inadequate for this kind of query/programming languages that we intend to develop.

The SBA solution relies on adopting a run-time mechanism of PLs and introducing necessary improvements to it. The main syntactic decision is the unification of PL expressions and queries; queries remain the only kind of PL expressions. For instance, in SBA there is no conceptual difference between expressions such as 2+2 and \((x+y)^{\ast}z\) and queries such as \(\text{Employee where Salary = 1000}\) or \((\text{Employee where Salary = (x+y)}\ast z).\text{Name}\). All such expressions/queries can be used as arguments of imperative statements, as parameters of procedures, functions or methods and as a return from a functional procedure. This approach is unique to SBQL – other PLs that involve querying capabilities always separate queries and expressions (especially if SQL is used as a query language). Even PLs that are claimed to be “integrated” with queries, such as PL/SQL or .Net C# + LINQ, assume some subdivision between queries and expressions. We consider this as an obvious sign of the impedance mismatch - a severe plague of current database applications. We believe that currently only SBQL consequently and in 100% has removed this plaque.

SBQL queries can be embedded within statements that can change the database or program state. Concerning this issue we follow the state-of-the-art standards known from majority of programming languages. Typical imperative constructs are creating a new object, deleting an object, assigning new value to an object (updating) and inserting an object into another object. We also introduce typical control and loop statements such as if...then...else..., switch, while loops, for each iterators and others. Some peculiarities are implied by queries that may return collections; thus there are possibilities to generalize imperative constructs according to this new feature.

Treating queries as programming language’s expressions requires an essential assumption concerning the semantics of queries. There was a lot of discussion in the literature concerning what queries have to return. For instance, an SQL query returns a tables that is a value rather than a collection of references to tuples (tuple identifiers). The ODMG standard assumes that queries return objects what is an obvious nonsense (see the discussion on the closure property) or some elliptic terminology. However, for some updating operators, in particular, for assignment, deleting and inserting, queries-expressions should return references to objects (i.e. object identifiers) rather than values or objects. Therefore during the development of SBA and SBQL we have returned to the classical notions of programming languages. In
SBQL from the very beginning we have assumed that queries never return objects but references to objects (in particular). This assumption is inevitable if one has to adopt queries as expressions of a programming language.

The presence of imperative constructs gives for the programmer some freedom if he/she should use queries or a sequence of programming statements. This freedom is first of all constrained by the conceptual modeling – the programmer should program his/her task as simply as it can be from his/her point of view. Usually declarative queries are much more comprehensive than sequences of commands. There is also a performance aspect: declarative queries are much more prospective for automatic query optimization than sequences of imperative statements.

In this chapter we describe in detail all the imperative constructs that we suggest to introduce in an integrated query/programming language. However, this is only a suggestion – we do not want to declare which construct can belong to SBQL and which cannot. Our goal is to develop the principle, not details, and to provide the basis for further examples presented in this book.

6.1 Declarations of Objects

In general, we make no difference between objects, as understood in object-oriented programming languages, and variables, as understood in classical programming languages. Both concepts are considered triples \(<i, n, u>\), as defined in the store models AS0, AS1, AS2 and AS3, where \(i\) is an internal unique object identifier, \(n\) is an object name (not necessarily unique), and \(u\) is an object value, which can be atomic (e.g. integer value), can be a pointer (an identifier of another object) or can be a set of objects. This definition is recursive and can define objects with any number of nested objects, with many hierarchy levels. In this description we avoid to use the term variable, naming variables objects too. Some people distinguish objects and variables according to their attitude to classes: objects are members of classes, unlike variables. However this distinction is secondary looking at the semantics of SBQL: we can simply assume that variables are objects for which classes are empty (i.e. there is no method that is associated to objects, hence creating such an empty class is omitted). Usually we assume that both objects and variables are associated with types that allow for strong compile-time type checking of the contexts in which objects/variables are used. Objects that have the nature of variables are associated with types only, while proper objects are associated with types that are invariants of classes.

Objects can be created in different environments. The most obvious environment is a local environment of the procedure or method where the given object creating instruction is put and executed. Such objects are volatile by definition, because their life is terminated when their procedures/methods are completed. Such objects are not available for any external bindings, e.g. from external procedures or methods. Objects can also be created within the module in which the given procedure/method is put and executed. By definition, such objects are shared among all the procedures or methods that are inside the given module. In other words, bodies of procedures or methods that are members of the given module can refer to all objects that are stored inside the module. Objects of a module can be also available for binding from other modules, under the condition that this is explicitly allowed by some module interface or export list (this is a form of encapsulation). In such a case usually references to objects are preceded by the module name (but this can be simplified in some cases).

Objects can also be persistent, that is, they are elements of a database. Persistent objects are also members of modules, but all their properties (values, OIDs, etc) outlive switching off the
application (and computer) that executes the given module. After restoring the application, its persistent objects become available in exactly the same form and value as before the switch off.

Objects can be shared among many parallel processes. To avoid inconsistencies, access to shared objects must be disciplined by ACID transactions. Usually it is assumed that persistent objects are always shared and volatile objects are never shared. In our view, sharing and persistence are orthogonal features. In particular, it is possible that persistent objects are not shared (they are available to only one application; hence the transactional semantics makes little sense). A typical case concerns a client application that needs to store persistently some results of calculations. And v/v, volatile objects that are stored within modules can be shared, hence should follow the transactional semantics. Hence some syntactic option should distinguish persistent objects and shared objects. Depending on a persistency/sharing model this option can belong to different definitions. We avoid the concept that some classes (hence its entire member objects) or types are qualified as persistent/shared. This concept makes impossible to define classes or types that are relevant to local, volatile and persistent objects, hence violates the principle of orthogonal persistence. Instead of that we can assume that proper flags (keywords) are assigned to modules rather than to classes.

Following strong typing, all objects that are to be created must be declared within a proper environment. Declaration has the form:

```
objectName: type [cardinality];
```

For instance:

```
x: integer;
```

An object named `x` with an integer value is declared; cardinality `[1..1]` is skipped.

```
Employee: record {
    name: string;
    salary: integer;
    job: string;
    worksIn: ref Company[0..1];
} [0..*];
```

A collection of objects named `Employee` is declared. The collection has any number of elements. The type is record (we avoid the term `structure` known from C/C++) having a subobject `name` (of string type), a subobject `salary` (of integer type), a subobject `job` (of string type) and an optional (cardinality `[0..1]`) subobject `worksIn` being a pointer to an object named `Company` (keyword `ref`). In our convention, typing of a pointer requires the name of the object that the pointer points to rather than the type of the object (unlike majority of programming languages). This decision is caused by database-orientation: in the database schemata object names are first class citizens while types are rather second-class and at high abstraction level are omitted at all.

The programmer is responsible to put such declarations in proper places, for instance, within the body of a procedure/method, within a proper module, etc. When declaration is valid, corresponding objects can be created by special instructions. When the lower cardinality of the type of an object is 1 (this also concerns the cardinality `[1..1]`, usually skipped) declaration of an object is equivalent to its creation with some default value. To be ready to use such an object must be initialized by an assignment.
6.2 Creating Objects

Creating an object requires determining two features: the state of a new object and its location (where the object is to be created). The second feature is usually related to object persistence - a newly created object is to be persistent when it has to be stored in a persistent environment (i.e. a database or persistent module).

Creation of an object can be determined by a very simple syntax:

\[
\text{instruction} ::= \text{create} \text{ query;}
\]

\[
\text{query} ::= \text{create} \text{ query;}
\]

It is assumed that the query being the argument of the create instruction should return all the information that is necessary to create an object or objects, in particular, names of objects and subobjects and their values. The concept of binder that was introduced in previous chapters is a very good mean for determining that. Hence it is assumed that the query returns a (perhaps) nested binder or several such binders. The instruction creates one or more objects according to the binders returned by the query, assuming that names of objects are determined by names of binders, values of objects are the same as the values of binders and nesting of objects exactly follows the nesting of binders. If the instruction is used as a query, it returns references (a bag of references) of a newly created object (objects).

### E.6.1 Create an Employee object.

**SBQL:**

```
create ("Doe" as name, 1000 as salary, "analyst" as job,
  (Company where compName = "ACME") as worksIn) as Employee;
```

The query being the argument of create returns the nested binder `Employee( struct{
  name("Doe"),
  salary(1000),
  job("analyst"),
  worksIn(iECME ) } ),` where \( i_{\text{ECME}} \) is the identifier of the ECME company object. This binder is used to create an object by a simple rule which assigns a new unique identifier to each (nested) binder.

Some SBQL users consider this syntax as a bit awkward in the context of creating instructions, thus one can invent a more friendly syntax (perhaps, only in this context). In the ODRA system the syntax \( q \text{ as } n \) can be substituted by \( n(q) \), hence the above example can also be written as:

### E.6.2 Create an Employee object.

**SBQL:**

```
create Employee (
  name("Doe"),
  salary(1000),
  job("analyst"),
  worksIn ( Company where compName = "ECME" )
);
```

In general, in this book we avoid the discussion on syntactic issues hence we will use the form E.6.1 rather than E.6.2. Both forms we consider semantically equivalent.

The create instruction must obey the strong typing system. In SBQL it is impossible to create a new object that does not conform to declared specifications. If the object is created as an element of a collection, the cardinality of the collection after creation is dynamically checked. Creating objects requires also altering the environment stack ENVS, which must be augmented by a corresponding binder (binders) in a proper stack section. If the created object is to be shared, it should follow the transactional semantics, i.e. it is locked till the given transaction will be committed. After committing, the object is to be available for other
transaction thus its binder must be propagated to all current environment stacks of particular applications.

Concerning the functionality and semantics of this instruction we need to give several comments.

In the *query* being the argument of this instruction returns several binders, the instruction creates as many objects as binders. In particular, if the query returns the empty bag, no object is created and this is not considered a failure. This rule is valid for any level of object nesting. For instance, if *Company* objects are declared as:

```plaintext
Company: record {
  compName: string;
  location: string [1..*];
  employs: ref Employee[0..*];
} [0..*];
```

then the instruction:

**E.6.3** Create a *Company* object.

**SBQL:**

```
create ( "ACME" as compName, 
  bag("Paris", "London", "Rome") as location, 
  (Employee where job = "analyst") as employs) as Company;
```

creates an object with three *location* subobjects and with some number of pointer objects *employs* leading to *Employee* objects.

Because the order of subobjects within an object is inessential, the order of binder constructs in the query is inessential too. For instance, E.6.4 is equivalent to E.6.1. This rule, however, can be changed if the defined query language has features that make the order of subobjects essential (this, for instance, is a desired feature of XML query languages).

**E.6.4** Create an *Employee* object.

**SBQL:**

```
create ( "analyst" as job, 
  (Company where compName = "ACME") as worksIn 
  "Doe" as name, 1000 as salary,) as Employee;
```

In general, a query being the argument of a create instruction can return binders with references as values. In such a case the instruction performs dereferencing (according to the specified type). The dereferencing is not performed in cases when the type specification provides a pointer object. For instance, the *Employee* type provides *worksIn* pointers and the *Company* type provides *employs* pointers (through *ref* keywords), hence the results of corresponding queries *Company where compName = “ECME”* and *Employee where job = “analyst”* are not dereferenced. In some cases *ref* keywords can be added before such queries, to explicitly show in the source code that dereferencing should not be performed and a pointer object is to be created. Using these keywords should be checked by the strong typing mechanism, to avoid programmers’ mistakes. E.6.5 is an equivalent to E.6.1, but there is more opportunity for strong type checking.

**E.6.5** Create an *Employee* object.

**SBQL:**

```
create ("Doe" as name, 1000 as salary, "analyst" as job, 
  ref(Company where compName = “ACME”) as worksIn) as Employee;
```

It may also happen that a query being the argument of the create instruction will return an identifier of a complex objects, but the type does not provide that a pointer is to be created. In
such a case the identifier is dereferenced according to a simple rule that maps the corresponding object into binders, by removing all object identifiers and retaining the nested structure of objects.

**E.6.6** Create a copy of the Doe’s object.

**SBQL:**

```sql
create (Employee where name = “Doe”) as Employee;
```

Assume that the Doe’s object is represented as the following nested triples:

```sql
<i1, Employee, {<i2, name, “Doe”>, <i3, salary, 1000>, <i4, job, “analyst”>, <i5, worksIn, iACME> }>
```

In this case the query `Employee where name = “Doe”` will return `i1`. Then, the `create` instruction requires dereferencing of `i1`:

```sql
deref(i1) = struct(name(“Doe”), salary(1000), job(“analyst”), worksIn(iACME))
```

Hence such an instruction is reduced to the case presented in E.6.1 and E.6.2. Dereferencing of a reference to a complex object is natural extension of the dereference of a reference to an atomic object (which return its value).

### 6.3 Locations of Created Objects

The location of an object is precisely determined by the strong typing system, thus in majority of cases there is no need to determine its locations by special syntax. During the static analysis phase (compilation) the type of a newly created object can be determined. This type is compared with type signatures that are stored at the static environment stack (S_ENVS – it will be explained in a chapter devoted to strong typing). The comparison starts from the top of the stack to its bottom and should determine a stack section having the corresponding signature. This environment is the proper one for creating the new object. If no corresponding signature is found on the stack, the statement is incorrect and the strong typing mechanism should display a typing error.

For instance, if the database contains specification of objects `Employee` and the local environment of some procedure contains a similar specification, then the instruction of creating an `Employee` object within this procedure will locate the object in the local environment rather than in the database. The local environment section is higher on the environment stack than the database section, hence it will be visited by the binding mechanism earlier that the database section.

If the environment is untyped (what we consider a very disadvantageous case) then the location of created objects must be determined by special syntax. In particular, in the LOQIS system [Subi90, Subi91] the keyword `create` can be augmented by keywords `permanent` or `temporary`. The first one denotes the database and the second one denotes a global user environment. If no such a keyword is specified, the new object is created within a local environment of the procedure that contains the instruction. A similar solution is also used in the ODRA system, but with a bit different meaning. SBQL implemented in ODRA is strongly typed. Keywords `permanent`, `temporal` and `local` are used for better understanding of programs by the programmer and present additional opportunity for strong type checking. There are, of course, a lot of other syntactic decisions concerning this aspect.

Because creating and deleting objects requires actions on an environment stack (perhaps on many environment stacks, if objects are shared among many applications or processes) the organization of environment stacks should support these operations. Actually, this optimization causes a bit different concept of environment stack construction. This new
concept has been implemented both in Loqis and in ODRA. In this concept binders that are to be stored at the environment stack are stored at the object store rather than the stack. The stack stores references to object store environments. This is illustrated in Fig. 6.1 (compare Fig. 2.2 and Fig. 3.4). The upper part of this figure presents the conceptual view on ENVS that is used to define the semantics of SBQL. The lower part of this figure presents the optimized version, where binders in some stack sections are substituted by an identifier of the proper object store environment. In this way operators create and delete that act on a store environment need not to update all the ENVS stacks that are currently present in all user sessions or processes. In Loqis and ODRA this organization is further optimized by inserting a section with proper binders (the result of the nested function) at the beginning of each complex object.

![Fig. 6.1. Conceptual and optimized environment stack](image)

The instruction can be augmented by an explicit clause determining where a new object is to be created. The syntax can be as follows:
instruction ::= create query within query;

query ::= create query within query;

In the construct \texttt{create }q_1\texttt{ within }q_2\texttt{ the query }q_1\texttt{ determines the state of the newly created object and the query }q_2\texttt{ determines the object inside which the new object is created}^{9}. Note that the above instruction is some generalization of the SQL insert instruction. As previously, the second syntactic form assumes that the construct returns references to newly created objects.

<table>
<thead>
<tr>
<th>E.6.7</th>
<th>Create an Address object within the Doe’s object.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td><code>create (&quot;Rome&quot; as city, &quot;Boogie&quot; as street, 88 as number) as Address within Employee where name = &quot;Doe&quot;;</code></td>
</tr>
</tbody>
</table>

The instruction must confirm to specification of types. The above instruction is correct only when the type of \texttt{Employee} includes an \texttt{Address} subobject in the form as presented above.

In principle, the above instruction can be considered as redundant, because it can be accomplished by two instructions: create and insert:

<table>
<thead>
<tr>
<th>E.6.8</th>
<th>Create an Address object within the Doe’s object.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td><code>create (&quot;Rome&quot; as city, &quot;Boogie&quot; as street, 88 as number) as Address; insert Address into Employee where name = &quot;Doe&quot;;</code></td>
</tr>
</tbody>
</table>

However, for two reasons this method would be inconvenient for programmer. First, it requires explicit specification of the type of object to be inserted and declaring a proper variable, which means two or more additional lines of code. Second, there could be problems with naming. For instance, declaring a local object \texttt{Address} could be awkward if the database would also contain objects \texttt{Address} – in this case binding to these database objects becomes impossible from the body of the current procedure code; hence some additional tricks could be necessary.

### 6.4 Deleting Objects

In SBQL we have assumed explicit deleting, as in SQL. Some authors assert that object programming languages should not allow for explicit deleting as such an operation is dangerous. It may cause that some pointers or references to deleted objects will become inconsistent (will became so-called dangling pointers). Hence an object is to be removed implicitly and automatically by the garbage collection mechanism in situation when the object is no more available (e.g. all pointers leading to it are nullified).

While these arguments have some value in case of lower-level languages, they are totally misleading in case of database programming languages. First of all, deleting an object has clear conceptual and business meaning. Conceptual modeling of programs is incomparably more important than any considerations concerning machine actions and features. Secondly, in the shared data environment the programmer may not enough rights and may have severe difficulties to recognize and remove/nullify all the pointers that lead to the object. There are also other arguments, related to database schema and operation on the schema. Summing up,

---

^{9} In the ODRA system this instruction has the syntax remaining the assignment operator:

\[
\text{lQuery} := \text{name(rQuery)};
\]

The left side determines where a new object must be created and the right side determines the state of the new object.
arguments that are valid for low-level programming are totally inadequate for business-level programming that we want to support.

The SBQL syntax for deleting objects is the following:

\[
\text{instruction ::= delete query;}
\]

We assume that the query returns references to objects that have to be deleted. The instruction physically deletes all of them, so they are no more available for further querying. If there are pointers that lead to the deleted objects they are deleted too or nullified. In ODRA it is assumed that each pointer from object A to object B is physically coupled with (invisible) backward pointer from B to A, hence this operation can be easily performed and always leaves the database in a consistent state. The instruction makes no difference concerning which kind of object is considered and on which object hierarchy level it is located – in the same way we may delete objects, attributes, views, stored procedures, etc. The instruction must conform to the type specification. Type failures can be detected statically (during compilation), e.g. deleting a subobject name within Employee, or dynamically (during runtime), in particular, violating declared cardinalities of a collection.

<table>
<thead>
<tr>
<th>E.6.9</th>
<th>Delete all objects of analysts.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>delete Employee where job = “analyst”;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E.6.10</th>
<th>Delete the Address subobject from the Doe’s object.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>delete (Employee where name = “Doe”).Address;</td>
</tr>
</tbody>
</table>

There are several peculiarities with this instruction:

Duplicates returned by the query should not cause failures. It may happen for various reasons that the query being the argument of the delete instruction will return duplicate references. If the object is deleted, an attempt to delete it next time may result in failure. Note that this requirement could be difficult to reduce to the case delete unique(query), where the unique function removes duplicates. As will be shown a bit later, duplicates may appear due to other rules and can be tangled within complex structures. Hence each instruction should collect identifiers of objects already deleted and check each next deletion if it is necessary or not.

Duplicates within pointers leading to deleted objects should not cause failures. If the list of references to objects to be deleted contains duplicates, the list of pointers that lead to the deleted object contains duplicates too. If the pointers are removed, the program should not fail.

The query may return binders with references rather than references alone. In such cases the deleting operation should not fail. It should remove objects according to their references being the values of the binders.

<table>
<thead>
<tr>
<th>E.6.11</th>
<th>Delete the Doe’s object.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>delete Employee as e where e.name = “Doe”;</td>
</tr>
</tbody>
</table>

A query being the argument of the delete instruction may return arbitrarily complex structure with references. This feature makes it possible to organize complex (cascade) deleting within one query.

<table>
<thead>
<tr>
<th>E.6.12</th>
<th>Delete the ECME company together with all the analysts that work for it.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>delete (Company where compName = “ECME”) as c</td>
</tr>
</tbody>
</table>
join (c.employs.Employee where job = "analyst") as a;

The query will return the bag of structures: bag struct(c(iComp1), a(iEmp1), struct(c(iComp2), a(iEmp2),... ) ). Note that in this case the query will return duplicates of c(iACME) binders, which should be handled without failures.

6.5 Assignment

The assignment operator can be introduced in the classical version:

\[
\text{instruction ::= query ::= query ;}
\]

Semantically, the assignment \textit{leftQuery ::= rightQuery} requires that the \textit{leftQuery} returns a reference to an object and the \textit{rightQuery} returns a value, which will be the new value of the object. If the \textit{rightQuery} returns a reference too, then the dereferencing operator is implicitly applied. The operator does not change the identifier of the object. In SQL the above operator is accomplished in the form of the update clause which also allows making more than one update within one statement. The same effect we can achieve by combining the assignment operator with the \textit{for each} operator (the example will be shown later). The SQL syntax we consider disadvantageous and not sufficiently orthogonal with other query language constructs, hence we do not use it. We also consciously avoid the use of the equality as an assignment operator (as in C/C++, Java, C#, etc.) using = as a comparison (following some hundreds years of the mathematical tradition and common elementary school teaching).

In case when an object referenced by the \textit{leftQuery} is specified with the cardinality [0..1], [0..*], etc. it may happen that the \textit{leftQuery} will return an empty result. This causes that the programmer for all objects specified by cardinalities with lower bound 0 must check the existence of the object before the assignment and to create if does not exist. This can be considered as an awkward feature. There is a possibility to change the semantics of the assignment operator to avoid this inconveniency, see the section devoted to the assignment to an absent object.

The assignment operator in the above syntax cannot be macroscopic. That is, the \textit{leftQuery} should return exactly one reference (if the assignment to absent object is not implemented) and the \textit{rightQuery} should return exactly one value. If these queries return bags it would be impossible to determine which value is to be assigned to a particular reference. If they return sequences, the situation is not better, because it could be very difficult to assure that the sizes of sequences are the same and the orders of references and values are correct. Moreover, in many cases parts of the queries must be repeated in left and right queries. It is much easier to use the operator \textit{for each} to achieve the effect of the macroscopic assignment.

In the Loqis system we also implemented the syntax

\[
\text{instruction ::= update query ;}
\]

with the assumption that the query returns a two-column table (a bag of two-element structures), where the first column contains references and the second column contains values. Then, for each row of this table the corresponding value is assigned to the corresponding reference. This form is a bit more powerful than the previous assignment form embedded within a \textit{for each} loop. However, its syntax appears to be awkward for large queries and the

\[10\] In SQL the character = is used as an equality and as an assignment in a single \textit{update} statement. Such an overloading of the operator is obviously error-prone.
advantage over the \textit{for each} construct is difficult to justify. In our examples we show the use of this construct and the reader will be able to assess if it has some advantages or not.

The same assignment operator can be used to update pointer objects and to update complex objects. We illustrate these cases by examples, see the schema from Fig.6.2.

![Database Schema](image)

Fig.6.2. A database schema

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>E.6.13</strong></td>
<td>For all programmers get rise of 5% (disregard optional jobs).</td>
</tr>
<tr>
<td>SBQL:</td>
<td>\texttt{for each Emp where job = “programmer” do { sal := sal * 1.05; }}</td>
</tr>
<tr>
<td>SBQL:</td>
<td>\texttt{update Emp where job = “programmer”. (sal, sal * 1.05);}</td>
</tr>
<tr>
<td><strong>E.6.14</strong></td>
<td>Increase the value of a local variable $x$ by 1.</td>
</tr>
<tr>
<td>SBQL:</td>
<td>$x := x + 1;$</td>
</tr>
<tr>
<td>SBQL:</td>
<td>update $x$, $x + 1;$</td>
</tr>
<tr>
<td><strong>E.6.15</strong></td>
<td>For all employees older than 40 get rise of 100 and change their job to engineer.</td>
</tr>
</tbody>
</table>
| SBQL: | \texttt{for each Emp where age > 40 do \{}
| & \texttt{sal := sal + 100;}
| & \texttt{job := “engineer”;}
| & \texttt{\}} |
| SBQL: | \texttt{update (Emp where age > 40). bag((sal, sal + 100), (job, “engineer”));} |

The second form is typologically incorrect (regarding the current SBQL implementation), as it joins within one bag a structure of two reals and a structure of two strings. It would be possible in languages without strong typing or with heterogeneous (variant) types. However, such cases are not recommended.
Note also that both statements are incorrect if *job* is absent. The full and correct version of this statement could be the following:

### E.6.16
For all employees older than 40 get rise of 100 and change their job to engineer (a fully correct version).

**SBQL:**
```
for each Emp where age > 40 do {
  sal := sal + 100;
  if not exists job then create “nothing” as job;
  job := “engineer”;
};
```

Such an *if..then* statement must be inserted in every case when lower cardinality of the object type is 0. This is considered awkward, thus can be improved by altering the semantics of the assignment; see the section devoted to the assignment to an absent object.

### E.6.17
For departments located in Paris change this location to Berlin.

**SBQL:**
```
for each (Dept.loc as dl) where dl = “Paris” do { dl := “Berlin”; }
```

Note that *loc* within *Dept*-s are collections and we have to update some elements of them. The assignment concerns the “auxiliary variable” *dl*, but the update will concern some *loc* subobject. In contrast to other query languages that we are aware of, SBQL auxiliary variables may bear references to objects as well.

### E.6.18
Move all programmers to the department managed by Doe.

**SBQL:**
```
for each (Emp where exists job) where job = “programmer” do {
  worksIn := Dept where (boss.Emp.name) = “Doe”; }
```

This statement updates pointers *worksIn* and *employs*. We have assumed (following ODRA implementation) that pointers *worksIn* and *employs* are bidirectional: updating of one of them causes automatic updating of its twin. We have also assumed that *worksIn* is typed as a pointer to a *Dept* object, hence *ref* before *Dept* is optional (we skip it). If the query language is not strongly typed and has no bidirectional pointers, the statement is more complicated:

### E.6.19
Move all programmers to the department managed by Doe.

**SBQL:**
```
for each ((Emp where exists job) where job = “programmer”) as p do {
  create (ref Dept where (boss.Emp.name) = “Doe”) as dd;
  delete Dept.employs where Emp = p; //removing old *employs* pointers
  create ref p as employs within dd.Dept; //new *employs* pointers
  p.worksIn := dd; //new *worksIn* pointers
};
```

A local object *dd* is used to store a pointer to the Doe’s department.

The statement can also be used to update complex objects. The semantics is very similar to the semantics of the *create* instruction. The left side of the updating instruction must return the reference to an object being update. The reference is not changed after updating. At the beginning, all subobjects of the object are deleted. Then, new subobjects are inserted according to the right side of the instruction. For a complex object it should contain a structure with binders; each binder is then changed into a subobject by generating a new unique object identifier. Names/values of binders are the names/values of created subobjects.
Get new data for the employee with eno = 223344.

**SBQL:**

```sql
(Emp where eno = 223344) := ( 
    "Lee" as name, 1980 as birthYear,
    ("London" as city, "Wall" as street, 15 as house) as address,
    223344 as eno, "programmer" as job, 2000 as sal,
    (Dept where dno = 789) as worksIn
); 
```

**SBQL:**

```sql
(Emp where eno = 223344) := ( 
    name( "Lee" ), birthYear( 1980 ),
    address( city( "London" ), street( “Wall” ), house( 15 ) ),
    eno(223344), job( “programmer” ), sal( 2000 ),
    worksIn( Dept where dno = 789 )
); 
```

The second syntactic form uses \( n(q) \) rather than \( q as n \). It seems to be better for reading.

While the identifier of an object being updated is not changed after updating, this does not concern its subobjects. In ODRA we assumed that all old subobjects are cancelled and new subobjects receive new identifiers. Although some identifiers of the subobjects can be retained, in general it may happen (due to optional and repeating subobjects) that some identifiers must be lost and some new identifiers must be generated. Semantically, it is difficult to imagine the situation in which the persistence of the identifiers of subobjects may have some meaning. However, if such situations exist, the identifiers of the subobjects can be retained as far as possible. This requires a bit more sophisticated implementation.

Our definition of the assignment operator allows also for cases when the right side query returns a reference to an object. Such a reference is then dereferenced according to a simple rule, as follows:

- For atomic objects \(<i, n, v>\): \( \text{deref}(i) = v \);
- For pointer objects \(<i_1, n, i_2>\): \( \text{deref}(i_1) = i_2 \);
- For complex object \(<i, n, \{<i_1, n_1, T_1>, <i_2, n_2, T_2>, \ldots, <i_k, n_k, T_k>\}>\), where \( T_i \) is an atomic value, an identifier or a set of objects:

  \[
  \text{deref}(i) = \text{struct}\{n_1(\text{deref}(i_1)), n_2(\text{deref}(i_2)), \ldots n_k(\text{deref}(i_k))\}.
  \]

Note that the definition is recursive and general. Below we present examples.
Dereferencing a reference to a complex object

Object:
\[
\text{\textless} i, \text{Emp}, \{
\text{\textless} i_1, \text{name}, \text{"Lee"}\>,
\text{\textless} i_2, \text{birthYear}, 1980\>,
\text{\textless} i_3, \text{address}, \{\text{\textless} i_4, \text{city}, \text{"London"}\>, \text{\textless} i_5, \text{street}, \text{"Wall"}\>, \text{\textless} i_6, \text{house}, 15\}\>,
\text{\textless} i_7, \text{eno}, 223344\>,
\text{\textless} i_8, \text{job}, \text{"programmer"}\>,
\text{\textless} i_9, \text{sal}, 2000\>,
\text{\textless} i_{10}, \text{worksIn}, i_{789}\}\>
\]

deref(i):
\[
\text{struct}
\begin{align*}
&\text{name( "Lee" ),} \\
&\text{birthYear( 1980 ),} \\
&\text{address( struct\{city( "London" ), street( "Wall" ), house( 15 ) \} ),} \\
&\text{eno(223344),} \\
&\text{job( "programmer" ),} \\
&\text{sal( 2000 ),} \\
&\text{worksIn(i_{789})}
\end{align*}
\]

For Nec assign all the data stored within the Doe’s object.

SBQL:
\[
(\text{Person where name = "Nec"}) := (\text{Person where name = "Doe"});
\]

Change the Poe’s address to the address of Doe (disregard optional addresses).

SBQL:
\[
(\text{Person where name = "Poe"}).\text{address} := (\text{Person where name = "Doe"}).\text{address};
\]

In some languages additional variants of the assignment are introduced: \( x += y \) denoting \( x := x+y \), \( x -= y \) denoting \( x := x - y \), etc. Such shortcuts are a bit controversial, although many programmers consider them useful.

The assignment operator meets also some problems with the substitutability principle. In the assignment \( X := Y \) both \( X \) and \( Y \) can be typed by type \( T \), but according to the substitutability \( X \) can return a reference to an object typed by some subtype \( T_1 \) of \( T \). This causes some doubts what to do with attributes that are present in \( T_1 \) but absent in \( T \). For instance, we can take the example E.6.22, but the person “Nec” is also an employee, hence his object may have also subobjects \( \text{eno} \), \( \text{job} \) and \( \text{worksIn} \) (and perhaps optional subobjects \( \text{sal} \) and \( \text{manages} \)).

According to our standard semantics of the assignment, in the first stage the internal of the object referenced by \( X \) is deleted, then the new internal is created, according to \( Y \). However, because “Doe” need not to have subobjects \( \text{eno} \), \( \text{job} \) and \( \text{worksIn} \), after the assignment the object referred by \( X \) cannot be typed as an \( \text{Emp} \) object. Hence the assignment has changed the type of the object, what may cause various inconsistencies in other parts of the program. For instance, the object is no more connected by a link \( \text{manages} \), hence some \( \text{Dept} \) object has no \( \text{boss} \) subobject, what is inconsistent with its type.

Perhaps there are simple solutions of this problem. For instance, we can cancel only those subobjects that are delivered by the type of \( X \), and leave without changes other subobjects. This means some complication of the semantics. Usually more complicated semantics is more difficult to implement and is more error prone. Perhaps some programming experience for such cases is required to propose a reliable and simple solution.
6.6 Inserting Objects

Insertion allows inserting an object into another object\(^{11}\). The syntax is as follows:

\[
\text{instruction} ::= \text{insert} \ query \ \text{into} \ query;
\]
\[
\text{instruction} ::= \text{insert} \ query \ \text{into} \ \text{module};
\]

Semantically, an instruction \(\text{insert} \ q_1 \ \text{into} \ q_2\) acts as follows. The result of \(q_1\) is a reference or a bag of references to objects (perhaps empty) and the result of \(q_2\) is a single reference to a complex object. The instruction moves the object(s) referenced by \(q_1\) inside the object referenced by \(q_2\). Objects referenced by \(q_1\) are physically moved without changing their structure and identifiers (i.e. they disappear in their old places). If \(q_1\) returns an empty bag, no action is performed and no error or exception is reported. If \(q_2\) returns an empty bag or the bag with more than one element the instruction is incorrect and causes an error or exception. Attempts to inserting an object into itself should also result in errors or exceptions. We also assume that both \(q_1\) and \(q_2\) can return binders with references as values; in such cases names of the binders are ignored.

In the second syntactic form \(\text{insert} \ q \ \text{into} \ \text{module}\) we assume that modules can be specified by different options that are unavailable as queries. For instance, modules can possess their specific names or can be specified by keywords such as local, global, persistent, etc. The semantics of this instruction is identical as the semantics described above.

In ODRA the syntax of the instruction reminds an assignment:

\[
\text{instruction} ::= l\ query :< r\ query;
\]

where \(l\ query ::= \ query\), \(r\ query ::= \ query\); an object (objects) referenced by \(r\ query\) is inserted into an object referenced by \(l\ query\).

Performing the inserting instruction should not violate type constraints, i.e. moving objects referenced by \(q_1\) should not cause violating types in their old place or violating the type of the object referenced by \(q_2\). Note that in our case types include cardinalities, which cannot be violated after performing the instruction.

The instruction should be perceived by the programmer as a physical moving of an object (objects) into another place without any change of their properties. However, this could be relaxed for cases when a volatile object is inserted into a persistent object or v/v. Depending on the method of assigning identifiers to objects (that may depend on its persistence status) the identifier of the inserted objects and the identifiers of all its subobjects can be changed.

For the AS2 store model this instruction should allow for inserting a new role into an object.

Similarly to creating instructions, the insert instruction should leave all the environment stacks that are currently operating by running processes or threads in a consistent state. Efficient implementation of this requirement requires changes in physical stack’s organization, as presented in Fig.6.1.

| E.6.24 | For each employee younger than 30 and having no job attribute, insert the subobject job with the value “assistant” (cf. Fig.6.2). |
| SBQL: | for each (Emp where age < 30 and not exists job) as e do { insert create “assistant” as job into e; } |

\(^{11}\) Note that the insert clause in SQL inserts new tuples into a table, hence from the conceptual point of view it corresponds to the create instruction explained above. SQL has no an inserting clause, as introduced here.
We follow the semantics of the create instruction which returns the reference of a newly created object. Note some typing peculiarity. Formally, because the object job is created in the current environment, we should assume that the type of the variable job is declared within it. However, because the object is created only for a moment, we can relax this requirement and skip declaring its type. The types will be finally checked after performing the entire instruction.

<table>
<thead>
<tr>
<th>E.6.25</th>
<th>Move the Paris location of the Sales department to the Ads department.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>insert (Dept where dname = “Sales”). loc as p where p = “Paris”</td>
</tr>
<tr>
<td></td>
<td>into Dept where dname = “Ads”;</td>
</tr>
<tr>
<td>Note</td>
<td>that the first query returns a binder named p with a reference to a loc subobject.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E.6.26</th>
<th>Move all the employees older than 65 from the current module to the Retired module.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>insert Emp where age &gt; 65 into Retired;</td>
</tr>
<tr>
<td>Note</td>
<td>that after the insertion all links worksIn/employs and manages/boss are not changed.</td>
</tr>
</tbody>
</table>

Sometimes the programmer wants to insert a copy of an object into a given object. The syntax could be as follows:

\[
\text{instruction ::= insert copy query into query;}
\]

\[
\text{instruction ::= insert copy query into module;}
\]

The copies of objects receive new unique identifiers.

<table>
<thead>
<tr>
<th>E.6.27</th>
<th>For all the persons with no addresses copy the address of Doe.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>insert copy (Emp where name = “Doe”).address into Emp where not exists address;</td>
</tr>
</tbody>
</table>

6.7 Changing Object Name

In many business applications objects can change their business role without changing their identities. A business role is frequently expressed by an object name. In terms of programming languages, change the name of a business objects requires change the name of an object or a programming variable. In majority of programming languages this is impossible because names of objects are second class citizens: they can be manipulated in a source code but cannot be manipulated during run time. The situation is different in databases, where all the names are available during run time, but usually there are no programming capabilities to change them. Additionally, changing object name can be constrained by its type.

In the current implementation of SBQL in the ODRA system all the names of objects (subobjects) are first-class citizen due to the assumed late binding of program entities. Hence there are no obstacles to change these names, according to the current business needs. This feature is so far not implemented in ODRA, but it is planned. Changing object name should be compatible with the declared types and the cardinalities of collections. The syntax can be as follows:

\[
\text{instruction ::= rename query to query;}
\]
Semantically, **rename \( q_1 \) to \( q_2 \)** implies the following action. Query \( q_1 \) returns references to objects to be renamed and query \( q_2 \) returns a string that will be the new name of the objects. The new name is assigned to the objects determined by \( q_1 \).

### E.6.28

| Ensure the database has collections Emp and SeniorEmp of the same type and class. Move all employees older than 50 to the collection SeniorEmp. |

**SBQL:**

```
rename Emp where age > 50 to “SeniorEmp”;
```

### 6.8 Control Statements

There are several good patterns of control statements that are developed in existing programming languages and we do not want to invent something radically new. Mostly we follow Java. We present them only for completeness of the definition and for using them in examples. In different versions of SBQL we have implemented the following constructs:

```
instruction ::= if query then programBlock

instruction ::= if query then programBlock else programBlock

programBlock ::= { Instructions } | { variables Instructions }

Instructions ::= instruction | instruction Instructions
```

In the statement **if \( q \) then \( p \)** query \( q \) should return a Boolean value. If \( q \) returns \( true \) the program block \( p \) is executed, otherwise no action is performed. In the statement **if \( q \) then \( p_1 \) else \( p_2 \)** if query \( q \) returns true then \( p_1 \) is executed, otherwise \( p_2 \) is executed. A **programBlock** can contain declarations of local variables; their scope is limited to the block.

```
instruction ::= case query do labelledInstructions endcase

instruction ::= case query do labelledInstructions else programBlock endcase

labelledInstructions ::= labelBlock | labelBlock labelledInstructions

labelBlock ::= label : programBlock

label ::= literal
```

This switching command is well known from other languages. In the syntax we assume that a **label** must be unique literal of the type determined by the **query**. After the **query** returns some value, it is compared with all the labels in the given **case** statement. If a **label** matches the query result the corresponding **programBlock** is executed. If no label matches the query, then no action is performed (the first syntactic variant) or the **programBlock** after **else** is performed (the second syntactic variant).

Other assumptions concerning labels are possible. In Loqis we implemented a variant where a label can be determined by any query (which should return an atomic value, which is to be compared with the result of the query after **case**). This variant gives a bit more possibilities for the programmer. However, because it is less explicit concerning the source code, it causes more opportunities for bugs.

Other group of control statements concerns organizing loops. In the current ODRA implementation these instructions follow Java, with the well-known semantics:

```
instruction ::= while query do programBlock

instruction ::= do programBlock while (query)
```
**E.6.29** For intervals [0, 99], [100, 199], [200, 299], …, [2000, 2099] print an interval and the number of employees earning a salary from the interval if the number is not zero.

**SBQL:**

```
for( x := 0; x <= 20; x := x+1 ) do {
  y: integer; z: integer;
  y := 100 * x;
  z := count(Emp where sal >= y and sal <= y+99);
  if z <> 0 then { print( y, y+99, z); }
}
```}

### 6.9 For each statement

The *for each* statement follows the semantics of non-algebraic operators of SBQL. It operates on the environment stack ENVS in a similar way as non-algebraic operators do. The operator processes collections of unknown sizes.

**instruction ::= for each query do programBlock**

The semantics of the instruction *for each q do p* is as follows. First, query q is evaluated; it can return a single element e, a bag of elements {e₁, e₂, …} or a sequence of elements sequence{ e₁, e₂, …}. A single element e is considered bag{e}. For each element e returned by q the following actions is performed:

- Program p is executed within this new state of the environment stack.
- All sections that are pushed at ENVS in the first step are popped.

If q returns a bag then the order of elements e is not determined, but all of them must be taken into account. If q returns a sequence, then the order of elements e is determined by this sequence. If an empty bag (sequence) is returned, the instruction does nothing and this case does not cause errors or exceptions.

In general, the instruction can be unsafe, e.g. when program p removes an object having a reference returned by q. In ODRA we do not prevent this inconsistency just in this place; in general, exception is raised if any operation is to be performed on non-existing object.

One can consider a syntactic variant of the *for each* instruction for the case when q returns exactly one element.

```
instruction ::= with query do programBlock
```
In this case the machine checks if query returns exactly one element ad causes typological error or exception otherwise. The semantics is exactly the same.

Unlike other proposals concerning for each statements, we do not introduce a special iteration variable (or variables). In SBQL such special variables are unnecessary. They can be introduced through as or group as operators as auxiliary names with the standard semantics.


| **E.6.30** | A variant of E.6.13 with an “iteration variable”. For all programmers get rise of 5% (disregard optional jobs). |
| **SBQL:** | `for each (Emp where job = “programmer”) as e do { e.sal := e.sal * 1.05; }` |

Two “iteration variables”:

| **E.6.30** | Print names of employees and names of their bosses in the alphabetic order according to the names of employees. |
| **SBQL:** | `for each Emp as e join (e.worksIn.Dept.boss.Emp) as b order by e.name do { print(“Employee ” + e.name + “ works for ” + b.name); newline(); }` |

| **E.6.31** | Increase the salary of Doe by 100 and change its job to engineer (assume there is exactly one Doe). |
| **SBQL:** | `with Emp where name = “Doe” do { sal := sal +100; if not exists job then { create “” as job; } job := “engineer”; }` |

6.10 Low-level Iterators

The presented above constructs for organizing program loops, including the for each statement, are sufficient to program almost all programming cases which require iterations. However, there are tasks when these capabilities are insufficient. The literature mentions an example of such tasks, namely, merging two sorted collections into one sorted collection. Assume that the collections are named C1 and C2 and the resulting collection is C. The algorithm can be represented in a pseudocode as follows:
C := empty_sequence;
e1 := get_first_element(C1);
e2 := get_first_element(C2);
while e1 <> null or e2 <> null do {
  if e2 = null or e1 < e2 then {
    C := C + e1; //concatenate C with e1
    e1 := get_next_element(C1);
  } else {
    C := C + e2; //concatenate C with e2
    e2 := get_next_element(C2);
  }
}

The algorithm cannot be coded by two nested for each statements, because a next element to be processed can be taken from the first or from the second collection, according to the result of comparison. For coding such tasks we need functions such as get_first_element, get_next_element and similar. In programming languages such functions are collected into a construct called iterator. An iterator is a data structure that stores an iterator state (e.g. a reference to a processed collection and a reference to a currently processed object) and some (usually small) collection of functions such as getFirst, getNext, getPrior, etc. If an iterator is to be applied to a result of a query, its state must contain this result. Except SQL, no programming language implements so general iterators. Usually iterators traverse stored collections, but generalization of them to collections derived by queries or views seems not to be a challenging task.

Low-level iterators imply a similar problem as for each statements in case when a collection being traversed by an iterator is augmented or reduced in its loop. Such modifications must be implemented with an extreme care, to avoid the situation when some element is to be processed two times or not processed at all.

In SQL such iterators are known as cursors. A big disadvantage of SQL cursors is that their names are global for the application, what makes difficulties e.g. with nesting iterators within functions that can be called from many parts of an application program (in particular, recursive functions). This can also be considered as a sign of impedance mismatch, because SQL interpreter (by definition) ignores the environment stack determining the scope for names. In integrated languages such as SBQL names and states of iterators should follow the stack-based discipline, similarly to names of all other local data structures in the stack-based approach. There are many good patterns of iterators in existing programming languages, e.g. Java, and we do not want to contribute to this state-of-the-art. In this chapter we give no syntax and semantics of iterators, considering them as further development of SBQL.

There are several other issues related to imperative statements in SBQL. They are well recognized in other object-oriented programming languages and are orthogonal to the stack-based approach that we want to explain on these pages. In ODRA we implemented exceptions taking the syntax and semantics from Java. Recently we have also implemented a prototype version of templates, taking the idea from C++. Implementation of other features is considered.

---

12 For this reason in the DBPL project [MRSS92] the author was forced to organize an own explicit stack of cursors that exactly duplicates the actions of the environment stack of Modula-2 (the programming language for DBPL in which SQL clauses were implicitly nested).
Procedures and Methods in SBQL

Procedures are the most important programming constructs for ensuring abstraction, encapsulation of code and program reuse. Practically all programming languages introduce procedures as a fundamental feature. Procedures encapsulate any complex calculations and hide their code from the programmer. They can be called from many places of applications and (in remote procedure call technologies) can be called from outside of the address space that a procedure is located in. Procedures can be parameterized explicitly, by parameters specified within their declarations, or implicitly, by an external environment that the body of a procedure can access and update (so-called side effects).

Depending on their properties procedures can be subdivided according to the following criteria (which can be orthogonally combined):

- **Proper procedures and functional procedures** (functions). Proper procedures belong to the category of program statements. They do not return a value, thus cannot be called in expressions or queries. Functional procedures return a value, thus can be called within expressions and queries. In majority of solutions functional procedures can be used as proper procedures, i.e. their calls can be considered statements. In this case the return value of a procedure is ignored. In some languages (e.g. C/C++) all procedures are functional ones, but concerning conceptual issues and strong typing this assumption is controversial.

- **Procedures and methods**. There are opinions that methods constitute “behaviour” of objects and they are activated by message passing. There are suggestions that message passing reminds communication in societies, thus presents a totally new paradigm of programming that contributes to parallel execution of programs. Such opinions are based on superficial observations and misleading associations. Parallel programming is a very important issue, but methods and message passing introduce to it exactly nothing. We follow the assumption that methods are ordinary procedures with a specifically constructed program environment (we explain it later) and message passing is equivalent to a procedure call (within special context that will be explained too). The most obvious difference between procedures and methods is their conceptual location. Procedures are located within program modules (sometimes within program blocks, e.g. Pascal), while methods are conceptually located within classes.

- **Procedures located within an application code and procedures stored in a database**. In traditional programming languages procedures are properties of an application code. Relational databases introduced a new kind of procedures, called stored procedures or database procedures that can be stored within a database. This issue is directly related to the phase of binding procedure names. Majority of programming languages assume early binding, i.e. binding during compilation and linking, before the program is executed. Stored procedures require late binding, i.e. binding during run time. The major advantages of early binding is much better performance and better opportunities for strong type checking. The advantage of late binding is flexibility and more possibility of generic programming, in particular, programming with linguistic reflection\(^\text{13}\).

- **Ordinary procedures and higher-order procedures**. Higher-order procedures can be parameterized by some programming entities that have properties of the source code, for instance, by types, by procedures, by classes, by lambda expressions, etc. Higher

---

\(^\text{13}\) Linguistic reflection is a property of a programming environment that makes it possible to dynamically generate a program that can be executed as a part of the generating program. The best-known language with linguistic reflection is LISP.
order procedures much increase flexibility and power, especially related to generic
programming. In contrast to linguistic reflection, which actually disables compile-time
strong type checking, higher-order procedures can be strongly typed as well. There are
various forms of type polymorphism proposed in different languages. This kind of
procedures much fascinated academicians and is practically ignored by industrial
professionals. There is old and perhaps already obsolete hope that higher-order
procedures will create a new quality in program development. Despite many attempts
(c.f. languages such as Scheme, ML, Haskell, F#) till now it is unclear if indeed they
present a new quality that can be sufficiently attractive for the common programmer.
Recently this kind of paradigm has been implemented within the LINQ project by
Microsoft Research (lambda expressions), but after some experience our impression is
that real advantage of this style of programming is difficult to identify. The obvious
disadvantage concerns query optimization, which is still much underdeveloped in
comparison to SQL and SBQL.

SBQL presents a new quality w.r.t. procedures and methods because we unify expressions
and queries. Hence queries can be parameters of procedures or methods, can be used within
return statements of functional procedures/methods and can be used within imperative
statements in bodies of procedures/ methods. Although considering the theory and strong
typing this is rather a well-recognized issue, none of the existing programming languages that
deal with queries (e.g. PL/SQL of Oracle, T-SQL of Sybase and Microsoft, ABAP of SAP,
SQL-2008 standard, C#LINQ) assume such radical treatment of queries in all contexts. We
show on examples that this SBQL feature indeed presents the new quality in application
program development.

Semantics of procedures is based on the environment stack mechanism that we have
introduced before for non-algebraic operators. The environment stack (known also as call
stack) was introduced in early 1960-ties for the programming language Algol-60. Thus the
history of this concept is almost as old as the idea of high-level programming languages. Our
original contribution is to use this stack (ENVS) to define non-algebraic query operators, such
as selection, projection, navigation, join, quantifiers, etc., see the section SBQL - Non-
Algebraic Operators. The semantics of queries based on ENVS is a sound substitution of very
limited, inconsistent, and usually mathematically incorrect semantics based on relational
algebra, relational calculi, functional approaches, object algebras, object calculi, monoid
comprehensions, F-logic and other concepts that were invented to deal with queries. Serious
critics of object algebras can be found in [Subi95c], but majority of the critics is relevant to
other formal approaches listed above. The stack-based approach that we describe on these
pages presents a specific conceptual construction that is motivated by the semantics of query
languages. The thorough description can be found in the section Environment Stack, Name
Scoping and Binding.

Usually developers of programming languages assume that procedures or methods can access
and update external entities, including application data, databases, files, external devices, etc.
directly, without use of parameters. This is known as side effects of procedures. Side effects
can be passive, that is, a procedure can access external entities, but cannot updated them, and
active, when a procedure can updated external entities. Currently popular programming
languages have no means to specify side effects of procedures. Originally, in the concept of
module that was proposed by David Parnas, side effects were specified in the form of a so-
called import list. This was accomplished in the programming languages Modula-2 and Eiffel.
Import lists were the components of types and access to external entities was strongly type
checked. There is a similar concept of import in languages such as Java, however, not as a
component of a procedure type. Interfaces in Java that specify signatures of procedures do not
contain information on side effects. This is sometimes criticized as a feature leading to errors. Methodologies such as Design by Contract provide specification of side effects of procedures (and a lot of other features such as preconditions and post conditions).

Syntactically, we distinguish declarations and calls (invocations) of a procedure/method. A declaration of a procedure consists of procedure name, formal parameters (single names) usually associated with their types, the type of its output (for functional procedures) and a procedure body where the programmer specifies a sequence of instruction that is to be performed. Within the body there are declarations of local objects or variables or sometimes other declarations (e.g. local procedures). A call of a procedure consists of procedure name and procedure actual parameters which are expressions. Types of the expressions must conform to the declared types of corresponding formal parameters. A procedure is encapsulated: the programmer that uses a procedure needs to know only its signature which consists of procedure name, formal parameters with their types and the type of a procedure output. The signature contains also some other information, for instance, concerning the method of parameter passing. For instance, there is a declaration of a procedure square:

```plaintext
square ( a: real): real { 
    counter: integer; x: integer;
    counter := 0; x := 1;
    while counter < 100 do { 
        x := (a/x + x)/2;
        counter := counter +1;
    }
    return x;
}
```

The signature of the procedure is

```
square( a: real): real
```

An example call of the procedure

```
square( 5 + z*t ) / 120
```

When a procedure or a method is called, a new section (so-called activation record) on the environment stack is pushed. The section contains three kinds of entities:

- Local procedure environment, i.e. all local objects/variables that are declared in the body of the procedure/method.
- Calculated actual parameters of the call. In some languages (e.g. C) calculated parameters are equivalent to variables, but this is not a rule.
- Return path, i.e. an address of the program code when the control should be passed after the procedure is terminated.

Sometimes a call implies pushing on the environment stack more sections. When a procedure or method is terminated, its activation record is popped. When a procedure is running, local environments of a procedure that called it are unavailable. This is due to the lexical scoping principle which assumes no bindings to entities of any environment that the programmer was not aware during writing a procedure. This is illustrated in Fig.7.1. Black sections denote entities unavailable for bindings. Bindings are performed from top of the stack to its bottom, as usually. In case of query languages the situation can be a bit more complex, because sections implied by procedure/method calls can be mixed with sections implied by non-algebraic operators. This will be presented later.
Procedures can be called from procedures without limitations, in particular, any recursive calls are allowed. Due to the stack-based semantics recursive calls are not extraordinary and in majority of languages recursive calls are not distinguished syntactically. However the following cautions should be observed:

- The size of the environment stack. It limits the depth of recursive calls, especially if the stack is organized in RAM.
- Within the procedure there should be a condition ensuring the end of recursion. Otherwise the recursive calls must result in failure, but if the stack is large, this may happen after long time. For this reason some languages limit the depth of recursion.
- Active side effects should be avoided or limited, because a next recursive call can overwrite the results of previous calls. Procedures that can be recursive without inconsistencies are called reentrant.

In SBQL we do not assume any limitations to the recursion. The programmer is responsible for avoiding stack overflows and for ensuring the end of recursion. Examples of recursive procedures in SBQL are presented in the chapter Recursive Queries in SBQL.

### 7.1 Parameters of Procedures

Procedures and methods can be parameterized. In contrast to mathematical functions, where a parameter is always evaluated to a value, parameters of procedures/methods can possess different semantics which influences their functionality and pragmatics of the use. Below we list several popular parameter passing methods. They can coexist within one language but usually must be syntactically distinguished.

#### 7.1.1 Call-by-value

This method assumes that the parameter is evaluated before the procedure code is started. The parameter should result in a value (perhaps a complex one). If evaluation of the parameter results in a reference of object/variable, the dereferencing operation is performed. There are two bit different treatments of the parameter within the procedure body. In languages such as
Pascal the parameter is not a variable, thus it is impossible to assign to it a new value. In C/C++ inside the body of the procedure the parameter is treated in the same way as a local variable. In some systems, e.g. CORBA, a call-by-value parameter in the declaration of a procedure and in its signature is preceded by the keyword *in*.

### 7.1.2 Call-by-reference

As before, the method assumes that the parameter is evaluated before the procedure code is started, but the evaluation must result in a reference to an object (variable). Within the procedure body the reference is used to update or delete the object. Hence the method assumes updates of an object from outside of the local procedure environment. Such a parameter is preceded by a special keyword: e.g. *var* in Pascal. In other systems (in particular built according to the CORBA standard) the parameter is preceded by keyword *out* or *inout*. There is a small difference in treatment of these two keywords: *out* means that the parameter is used for output only, while *inout* means that the parameter can be used for input and for output. This difference can be checked by the strong typing system, which should forbid the dereferencing operation for *out* parameters.

### 7.1.3 Strict-call-by-value

As for call-by-value, but no dereferencing operation is performed. A reference is treated in the same way as a value, hence the method can be applied without a special syntax as *call-by-value* or *call-by-reference*. The method implies less opportunities for strong type checking, but lack of a special syntax for distinguishing kinds of parameter passing is an advantage. The method is used in such languages as C. It has some meaning for query languages if the programmer wants to use a parameter which is evaluated to a bag of structures joining values and references.

### 7.1.4 Call-by-value-return

As for call-by-reference, but a copy of a referenced object within local procedure environment is created. All operations on the referenced object are then performed on this local copy. At the end of the procedure, the value of the copy is sent as the value of the original object. The method has a great meaning in distributed environments, as it minimizes the number of connections to remote objects. However, the method is not semantically clean, if two or more call-by-value-return parameters refer to the same object (what may happen due to nested loops, processing bags, etc.) In this case two or more copies of the same object will be created; all copies can be updated, but only updates of the copy that is sent as the last will be indeed recorded in the original object. Other updates will be lost, which as a rule means an error in the program.

### 7.1.5 Call-by-name

The method assumes that the parameter is not evaluated before the procedure body is started. Instead, an expression being the parameter is identified. A pointer to the expression code is passed to the body of the procedure. When the control meets this pointer within the procedure body, the expression is evaluated (in the caller environment). The method has some advantage: it avoids evaluation of the parameter if the control does not meet the above
pointer. However, the method may require evaluation of the parameter many times. Moreover, each evaluation can return a different result (if the caller environment is changing), thus the method requires attention from the programmer. The method (implemented in Algol-60) is considered historical and obsolete, but the come-back can be considered in the context of query languages. If the environment necessary for evaluation of the parameter is not changed (e.g. it is a database) then the method can be implemented in such a way that each occurrence of the parameter within the procedure body is macro-substituted by the expression communicated as a parameter. This would give more opportunities for query optimization (rewriting, indices, etc.), because a query being the parameter is joined with a query that uses the parameter. We illustrate this method in example E.7.1. The method is similar to the method of processing views known as query modification (explained later).

<table>
<thead>
<tr>
<th>E.7.1</th>
<th>C.f. Fig.6.2. Procedure MyEmp returns references to Emp objects communicated as the first parameter having names communicated as a second parameter. Assume there is an index for name and no index for sal.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>MyEmp ( who: EmpType[0..<em>]; myNames: string[0..</em>] ) : EmpType [0..*] { return who where name in myNames; }</td>
</tr>
</tbody>
</table>

Call the procedure MyEmp for employees earning more than 1000 and for names “Doe”, “Poe” and “Lee”.

| SBQL: | print( MyEmp( Emp where sal > 1000; bag{“Doe”, “Poe”, “Lee” } ) ); |

Note that the call cannot be optimized too. However, when the call-by-name method is applied, the query who where name in myNames is rewritten to:

(Emp where sal > 1000) where name in bag{“Doe”, “Poe”, “Lee”}

This query can be further rewritten to the semantically equivalent form:

(Emp where name in bag{“Doe”, “Poe”, “Lee”}) where sal > 1000

The query can be optimized by using the index for name.

7.1.6 Call-by-need

Sometimes the method is called lazy evaluation. Its assumptions are similar as for call-by-name, but the parameter is evaluated only once, first time when the control meets the pointer to an expression being the parameter. After evaluation, its result is stored within the local procedure environment and then reused each time when the control meets the pointer an expression. It has some performance advantage over call-by-name, but in comparison it gives fewer opportunities for query optimization that we have mentioned for call-by-name.

If we assume that parameters can be queries, the situation concerning parameter passing methods is changed. A query can return a complex value which consists of nested bags, sequences and structures. Moreover, the value can contain values and references. As we can see from example E.7.1, some parameter passing methods can be much better to optimize than the traditional methods. For this reason the decision which method can be chosen for parameters being queries requires more research and experience.
7.2 Syntax and Semantics of SBQL Procedures

In SBQL there is little distinction between procedures and functional procedures. Semantically, the only difference is that a functional procedure name is eventually used to store the output from the procedure. Thus, an invocation of a functional procedure can be treated as a query and can be nested as a part of other queries. In principle, a value of any type (including nested structures, bags, sequences, etc.) can be allowed as an output from a functional procedure. Any limitations in this respect violate the orthogonality principle and are not justified by implementation (which is equally easy for any type). A non-functional procedure cannot be a part of a query because no value is assigned to its name. Typing of procedures and functional procedures is a bit different: the type of a proper (non-functional) procedure does not contain a return type. All procedures can call procedures without limitation; arbitrary recursive calls are allowed with no syntactic distinction. A procedure is uniquely identified by its name. Overloading of procedure names is not supported (but this of course can be changed, e.g. in the spirit of Java). The result of a functional procedure is typed similarly to other programming languages. Each functional procedure can also be used as a proper procedure; in this case its return value has no meaning (is dropped).

A procedure declaration consists of a procedure name, a typed parameter list, a type of its output and a list of imperative SBQL statements.

```
proc_declaration ::= name ((formal_par_list)) [: return_type] { [statements] }
formal_par_list ::= par_declaration [; parameter_list]
statements ::= instruction [ statements ]
instruction ::= …any instructions defined previously and procedure calls…
instruction ::= return ; | return query ;
par_declaration ::= [par_transmission_method] par_name : type [ cardinality ]
par_transmission_method ::= in | out | inout | strict | macro | …
```

Procedure parameter declaration syntax determines its name, type and cardinality. For a given procedure names of parameters must be unique. If the cardinality is not specified, the default cardinality [1..1] is assumed. In the procedure signature parameter declarations are separated by semicolons. Instruction return concerns proper procedures and causes immediate passing control to the caller of a procedure. It need not necessary if execution of the body of the procedure is terminated naturally. Instruction return query causes the same, but with storing the result of query on top of the QRES stack. This instruction must occur at least once within a functional procedure. In this case instruction return without query is not allowed and ending the procedure without return query is not allowed too. Parameter transmission methods are distinguished syntactically by keywords. The default method (no keyword) is equivalent to in and denotes call-by-value. Keywords out and inout denote two variants of call-by-reference. Keyword strict denotes strict-call-by-value. Parameter macro denotes a variant of call-by-name in which the parameter is treated as a macro-definition. Other parameter passing methods can also be defined, depending on the research and experience.

Procedure invocation syntax is the following:

```
proc_invocation ::= name [(actual_par_list)]
actual_par_list ::= parameter [; actual_par_list]
```

14 In ODRA only strict-call-by-value is implemented, hence no keyword precedes a parameter declaration.
parameter ::= query | par_name : query

Syntactically, a call of a procedure consists of its name and parameters. If the list of parameters is empty, it is allowed to drop parentheses (as in Pascal). Lack of parentheses may have some meaning; in particular, it allows abstracting from the distinction between procedure or method call and binding to stored objects. For instance, if the class of objects Person has the functional method age, then the query Person where age > 25 looks more naturally than Person where age() > 25. For better legibility and reducing possible errors each actual parameter can be preceded by the corresponding name of a formal parameter. If all actual parameters are preceded by names of formal parameters, then the order of actual parameters is inessential. Additionally, if a mechanism of parameter default values would be implemented (we explain it later) then some or all parameters can be dropped. Otherwise, the order and the number of actual parameters must correspond to the order and the number of formal parameters.

Inferred types of actual parameters must conform to declared types of formal parameters, the conformity is determined by the substitutability principle or by some implemented forms of \textit{ad hoc} conformity (e.g. the \texttt{integer} type can be used when the \texttt{real} type is expected). So far we do not determine precise rules of \textit{ad hoc} conformity, as the typing system of SBQL can differ in details (one of the variants is implemented in ODRA, but this is not a standard).

Semantics of a procedure call follows the semantics explained above. Parameter transmission methods \textit{call-by-value}, \textit{call-by-reference} and \textit{strict-call-by-value} require evaluation of the actual parameter before the control is passed to the body of the procedure. The evaluation of a parameter results by a new section on the query result stack QRES. Then, a new section (activation record) is pushed on the environment stack; the section consists of actual parameters (represented as binders), local procedure objects (represented as binders too; the corresponding objects are stored in a special part of a volatile store) and return path (represented as an address of a compiled bytecode). Then, values of the parameters are popped from QRES. After this section is created, the control is passed to the body of the procedure, i.e. sequence of instructions (statements). Instructions from the body are executed; the order is determined by their sequence and by imperative control statements explained in the chapter \textit{Imperative Constructs of SBQL}. When the control meets the end of the sequence of instruction or a \textit{return} statement, the procedure is terminated. Termination means the following actions:

- For functional procedures and methods the result of the query within the \textit{return} statement is pushed at the top of QRES.
- The return path is stored within some auxiliary variable.
- All local objects that were created for this procedure are removed.
- The environment stack ENVS is popped – the section that was pushed at the stack when the procedure was started is removed.
- The control is passed to the caller code according to the return path.
8 Recursive Queries in SBQL

There are many important tasks that require recursive processing. The most widely known is Bill-Of-Material (BOM), a part of Materials Requirements Planning (MRP) systems. BOM acts on a recursive data structure representing a hierarchy of parts and subparts of some complex material products, e.g. cars or airplanes. Typical BOM software processes such data structures by proprietary routines and applications implemented in a classical programming language. Frequently, however, there is a need to use *ad hoc* queries or programs addressing such structures. In such cases the user requires special user-friendly facilities dealing with recursion in a query language. Similar problems concern processing and optimization tasks on genealogic trees, stock market dependencies, various types of networks (transportation, telecommunication, electricity, gas, water, and so on), etc. Recursion is also necessary for internal purposes of computer systems, such as processing recursive metadata structures (e.g. Interface Repository of the CORBA standard), software configuration management repositories, hierarchical structures of XML or RDF files, and others.

In many cases recursion can be substituted by iteration, but this implies much lower programming level and less elegant problem specification. Iteration may also cause higher cost of program maintenance since it implies a clumsy code that is more difficult to debug and change.

Despite importance, recursion is not supported in SQL standards SQL-89 and SQL-92. Beyond the standards, it is implemented (differently) in relational database management systems, in particular, in Oracle and DB2, in the form of transitive closure and linear recursion. The newer standards SQL-3, SQL-99, SQL-2003, SQL-2008 introduce both transitive closure and deductive rules *a la* Datalog [Ceri89]. Unfortunately these standards are very huge and eclectic thus there are doubts among database professionals if they will ever be fully implemented, especially concerning such advanced properties.

The ODMG standard for OO databases does not mention corresponding facilities. This will obviously lead to extensive programming in programming languages such as C++, Java and Smalltalk. Recursion is considered a desirable feature of XML and RDF-oriented query languages, but current proposals and implementations do not introduce corresponding features to the corresponding query languages or introduce them with limitations.

The possibility of recursive processing has been highlighted in the deductive databases paradigm, notably Datalog. The paradigm has roots in logic programming and has several variants. Some time ago it was advocated as a true successor of relational databases, as an opposition to the emerging wave of object-oriented databases. Despite high hype and pressure of academic communities it seems that Datalog falls short of the software engineering perspective. It has several recognized disadvantages, in particular:

- Lack of efficient methodology supporting the developers of applications in transition from business conceptual models to Datalog programs. For real applications an average developer or programmer has no idea how to formulate Datalog rules in response to typical results of analysis and design processes (stated e.g. in UML).
- Although Datalog is claimed to be a universal query language, its real application area is very limited to niche applications requiring some “intelligence” expressed through syllogisms and recursive rules.
- Limitations of data structures that Datalog deals with. Current object-oriented analysis and design methodologies as well as programming languages and environments deal with
much more sophisticated data structures (e.g. complex objects with associations, classes, inheritance, etc.) than relational structures that Datalog deals with. Complex data structures allow one to get the software complexity under control.

- „Flatness” of Datalog programs, i.e. lack of abstraction and encapsulation mechanisms, such as procedures, views or classes. This flaw means no support for hierarchical decomposition of a big problem to sub-problems and no support for top-down program design and refinement and encapsulation of problem details.
- Datalog is stateless thus it gives no direct possibility to express data manipulation operations. Majority of applications require update operations, which are possible to express in Datalog only via side effects, with no clear formal semantics.
- Datalog, in principle, does not separate data and programs. Both are “rules” in the same abstraction frame. However, the fundamental differences between these two aspects concern the following factors: design of programs (databases are designed before programs that operate on them), client-server architecture (databases are on servers, programs are usually on clients); structural constructs (data structures are specifically constructed and constrained, especially in object-oriented databases; this does not concern programs); updating (data can be updated, while it makes little sense for programs). These fundamental differences cause that the unification of data and programs must meet severe limitations.
- Datalog implies significant problems with performance. Current optimization methods, such as magic sets, do not seem to be sufficiently mature and efficient.

Thus Datalog applications supporting all the scale that databases require are till now unknown. Nevertheless, the idea of Datalog semantics based on fixed-point equations seems to be very attractive to formulate complex recursive tasks. Note however that fixed-point equations can be introduced not only to languages based on logic programming, but to any query language, including SQL, OQL and XQuery. Below we show how to incorporate fixed point equations in SBQL. In contrast to Datalog, we assume that such a feature can be used by the SBQL programmer only when needed rather than as the major programming paradigm.

Besides transitive closures and fixed-point equations there are classical methods for recursive processing known from programming languages, namely recursive functions (procedures, methods). In the database domain a similar concept is known as recursive views. Integration of recursive functions or recursive views with a query language requires generalizations beyond the solutions known from typical programming languages or databases. First, functions have to be prepared to return bulk types that a corresponding query language deals with, i.e. a function output should be compatible with the output of queries. Second, both functions and views should possess parameters, which could be bulk types compatible with query output too. Currently very few existing query languages have such possibilities, thus using recursive functions or views in a query language is practically unexplored.

This chapter discusses three different approaches to recursive queries:

- Transitive closures, i.e. special query operators for processing recursive data structures. We show that the SBQL transitive closures have the power that was not even considered in other query languages. At the same time, their semantics and implementation is as easy as possible.
- Least fixed point equations (fixpoint equations, for short), which allow for recursion in the spirit of Datalog programs, but without the mentioned above disadvantages of Datalog;
- Recursive procedures and views. This is a traditional facility in programming language to express recursive computations. However, in SBQL these facilities are integrated with
queries. In particular, SBQL procedures accept queries as parameters and may return any bulk output (hence invocations of them can be combined with queries).

The first approach is implemented in a few industrial commercial systems with severe limitations [Piec04]. Two next approaches are implemented so far in research prototype systems only, with limitations too. Till now no system implements all three approaches in a sufficiently robust version, thus comparison of them from the software designer and programmer point of view is unfeasible. Moreover, all widely known implementations concerns relational database systems which are successful, but with well-recognized disadvantages such as impedance mismatch and poor conceptual modeling. There are few theoretical proposals concerning recursive queries for object-oriented and XML-oriented systems, but implementations are not widely known or are very limited. Only SBQL implements all three approaches in sufficiently robust and simple versions.

Recursive queries can imply performance problems. There are several typical optimization methods related to recursive queries, such as factoring out independent subqueries, indices, or caching (materializing) results of queries to utilize them during evaluation of next (identical) queries. For fixpoint equations there are special methods know as magic sets. In the chapter we briefly discuss the above issues.

8.1 Transitive Closures

A transitive closure in SBQL is a non-algebraic operator having the following syntax:

\[ \text{query ::= query close by query} \]
\[ \text{query ::= query leaves by query} \]

The query \( q_1 \) close by \( q_2 \) is evaluated as follows. Let final_result be the final result of the query and union be the bag union. The pseudo-code accomplishing abstract implementation of \( q_1 \) close by \( q_2 \) is the following:

\[
\text{final_result} := \text{result_of}(q_1);
\]
\[
\text{for each } r \text{ in final_result do } \{
\text{push nested}(r) \text{ at top of ENVS;}
\text{final_result} := \text{final_result union result_of}(q_2);
\text{pop ENVS;}
\}
\]

Note that each element \( r \) added to final_result by \( q_2 \) is subsequently processed by the for each command. The above operational semantic can be described in the denotational setting as the least fixed point equation (started from \( \text{final_result} = \text{empty_bag} \) and continued till fixpoint):

\[
\text{final_result} = q_1 \text{ union final_result}.q_2
\]

where the semantics of dot is identical with the dot operator in SBQL. Similarly, the semantics can be expressed by iteration (continued till \( \text{result_of}(q_2) = \text{empty_bag} \)):

\[
\text{final_result} = q_1 \text{ union } q_1.q_2 \text{ union } q_1.q_2.q_2 \text{ union } q_1.q_2.q_2.q_2 \text{ union } ....
\]

Naive implementation of the close by operator is as easy as the implementation of the dot operator. Note that if \( q_2 \) returns a previously processed element, an infinite loop will occur. Checking for such situations in queries is sometimes troublesome and may introduce unnecessary complexity into the queries. Another operator close unique by has been introduced to avoid infinite loops due to duplicates returned by \( q_2 \).
The query $q_1$ leaves by $q_2$ is evaluated as $q_1$ close by $q_2$, but in the result only leaves of the transitive closure graph are left. The semantics can be expressed as follows:

$$\text{final_result} := \text{result_of}(q_1);$$

for each $r$ in final_result do {
  push nested($r$) at top of ENVS;
  temporary_result := result_of($q_2$);
  final_result := final_result union temporary_result;
  if temporary_result <> bag() then final_result := final_result minus \{ $r$ \};
  pop ENVS;
}

$\text{bag()}$ stands for empty bag. As $q_1$ and $q_2$ can be any queries, simple or complex, the relation between elements which is used for transitive closure is calculated on the fly during the query evaluation; thus the relation need not be explicitly stored in the database.

Fig.8.1. A sample recursive database schema

Fig.8.1 depicts a simple database schema used in our examples. It is a description of parts, similar to descriptions used in Bill of Material (BOM) applications. Each Part has a name. The part may be either a Detail or Aggregate. Each detail has attributes cost and mass (its cost and mass). Each Aggregate has attributes assemblyCost and assemblyMass which represent the cost of assembling this aggregate and mass added to the mass of its components as the result of the assembly process. Aggregates have one or more Component subobjects. Each Component has the attribute amount (number of components of specific type in a part), and a pointer object leadsTo leading to a Part object.

The following SBQL query with a transitive closure over this schema finds all components of a part named “engine”.

**E.8.1** All components of a part named “engine”.

**SBQL:**

$$(\text{Part where name = "engine"}) \text{ close by Component.leadsTo.Part}$$

This query first selects parts having the attribute name equal to “engine”. The transitive closure relation is described by the subquery Component.leadsTo.Part. It returns all Part objects which can be reached by the pointer leadsTo from already selected objects.
This query takes advantage of an assumption made in the AS1 model. After reaching an object by its name all its attributes are accessible, not only those of the class whose name has been used to retrieve the object. The assumption allows us to simplify queries by removing the necessity to use cast operators.

If an object does not have the queried attribute (because it belongs to a subclass which does not have this attribute or the attribute is optional), the empty collection is returned as the result of binding of the name of this attribute. In this query, the subquery Component.leadsTo.Part is evaluated for Part objects which can be either Detail or Aggregate. In case of Aggregate, the name Component is properly bound and returns the collection of the Component subobjects. In case of Detail, the name Component cannot be bound and returns the empty bag.

Assuming there is no access from a Part object to Aggregate and Detail attributes the query can be formulated with a cast:

**E.8.2**  
All components of a part named “engine”.

**SBQL:**  

```sql
((Part where name = "engine") as e) close by ((Aggregate)e). Component.leadsTo.Part
```

One of the basic BOM problems, i.e. finding all components of a specific part, together with their amount, may be formulated using the transitive closure as follows:

**E.8.3**  
Find all components of a specific part, along with their amount required to make this part.

**SBQL:**  

```sql
((Part where name="engine"), 1 as qty) close by Component.(leadsTo.Part), (qty*amount) as qty)
```

The query uses a named value in order to calculate the number of components. The number of parts the user wants to assemble (in this case 1) is named qty and paired with the found part. In subsequent iterations the qty value from parent object is used to calculate the required amount of child elements. It is also named qty and paired with the child object.

The above query does not sum up amounts of identical sub-parts from different branches of the BOM tree. Below we present a modified query which returns aggregated data – sums of distinct components from all branches of the BOM tree:

**E.8.4**  
Find all components of a specific part, along with their amount required to make this part, summing quantities of identical parts.

**SBQL:**  

```sql
(((Part where name="engine") as x, 1 as qty) close by x.Component.(leadsTo.Part) as x, (qty*amount) as qty)) group as allEngineParts.

((distinct(allEngineParts.x) as y).

(y, sum(allEngineParts where x=y).qty)))
```

This query uses grouping in order to divide the problem into two parts. First, all the components named x, along with their amounts named qty are found. The pairs are then grouped and named allEngineParts. The grouped pairs are further processed by finding all distinct elements and summing the amounts for each distinct element.

This query could be further refined in order to remove all aggregate parts (so only the detail parts will be returned). There are many ways to accomplish this goal. One of them is to use the operator leaves by in place of close by. The operator leaves by returns only leaf objects, i.e. objects which do not result in adding any further objects to the result bag:
### E.8.5
Find all details of a specific part, along with their amount required to make this part, summing quantities of identical details.

**SBQL:**

```sql
(((Part where name="engine") as x, 1 as qty)
  leaves by x.Component.((leadsTo.Part) as x, (qty*amount) as qty))
  group as allEngineDet).
  ((distinct(allEngineDet.x) as y).
   (y, sum((allEngineDet where x=y).qty)))
```

The other way to sort the aggregates out of the result of the previous query is to use a cast. The cast operator takes a collection of objects and returns only these items of it which belong to the given class. The cast applied to a single object returns this object if it belongs to the class; otherwise it returns the empty bag. Here is a query with the same retrieval goal but written using a cast.

### E.8.6
Find all details of a specific part, along with their amount required to make this part, summing quantities of identical details.

**SBQL:**

```sql
(((Part where name="engine") as x, 1 as qty)
  close by x.Component.((leadsTo.Part) as x, (qty*amount) as qty))
  group as allEnginePar).
  ((distinct((Detail) (allEnginePar.x)) as y).
   (y, sum((allEnginePar where x=y).qty)))
```

Here the result of the subquery whose result is further named `allEngineDet` is first cast to the class `Detail`. This cast drops all objects which do not belong to this class.

Although the complexity of the SBQL solution is still high, SBQL supports facilities to manage the complexity. In this case the grouping operator allows decomposing the problem into easier subproblems.

SBQL queries may be used to perform even more complex tasks. The query below calculates the cost and mass of the part named “engine”, taking into account cost and mass of each engine part, amount of engine parts and cost and mass increment connected with assembly. This task has been used in [Atki87] as an example of lack of power and flexibility of currently used query languages. In SBQL the task can be formulated with no essential problems:

### E.8.7
Calculate the cost and mass of the part named “engine”, taking into account cost and mass of each engine part, amount of engine parts and cost and mass increment connected with the assembly.

**SBQL:**

```sql
(((Part where name="engine") as x, 1 as qty)
  close by x.Component.((leadsTo.Part) as x, (amount*qty) as qty))
  group as allEngineParts).
  (allEngineParts.(Detail) x as d). ((qty * d.cost) as c, (qty * d.mass) as m)
  group as CostMassIncreaseD,
  allEngineParts.(Aggregate) x as a).
  ((qty*a.assemblyCost) as c, (qty*a.assemblyMass) as m)
  group as CostMassIncreaseA).
  ((sum(CostMassIncreaseD.c) + sum(CostMassIncreaseA.c)) as engineCost,
   (sum(CostMassIncreaseD.m) + sum(CostMassIncreaseA.m)) as engineMass)
```
Such a typical BOM task cannot be formulated so comprehensively as a single query in other query languages. For example, this query formulated in SQL-99 exceeds 100 lines of code and is totally illegible. Furthermore, some of its subqueries have to be repeated several times in its text.

SBQL can also perform quite complex calculations (not even considered in other query languages). Let the object named $a$ store some non-negative number. The query below calculates approximation of the square root of $a$, using the fixed point equation $x = (a/x + x)/2$, starting from $x=1$ and making 50 iterations:

**E.8.8** Calculate the approximation of the square root of $a$.  

**SBQL:**

```
((1 as x, 1 as counter)
leaves by (((a/x + x)/2 as x, counter+1 as counter) where counter≤50)).
(x where counter = 50)
```

In this example we use a kind of internal variable `counter` as a counter of iterations.

Another example of a recursive task:

**E.8.9** Calculate 50 Fibonacci numbers according to the recursive definition: $F(0) = 0; F(1) = 1; F(n) = F(n-2) + F(n-1)$. The output should contain the argument of $F$ and the result of $F$.

**SBQL:**

```
((0 as arg, 0 as fib, 1 as nextfib)
close by ((arg+1) as arg, nextfib as fib, (fib + nextfib) as nextfib)
where arg < 50)).
```

Although the calculations remind programming, we underline that we are still using query operators only. The examples show the power of recursive queries in SBQL so far not even considered in other query languages.

Cycles in the queried graph can be easily dealt with by means of another variant of the close by operator – close unique by. This variant removes duplicates on the fly after each closure iteration; thus cycles do not imply infinite loops. Another variant of the close by operator is the leaves unique by operator. It is a combination of the two previous variants. It returns only leaf objects, while preventing problems with cycles in graphs. Note that cycles in the queried graph do not have to be an effect of database inconsistency. It is easy to imagine a database which contains a graph with cycles that is consistent with the real world situation (see Fig.8.2) .

![Diagram](image)

**Fig.8.2.** A sample structure for the examples of an application of close unique by

A Company has name and may own shares in other Companies (in this example we abstract from the amount of shares). Other companies in turn may own shares in further companies; and so on. It is possible for a company (call it “ACME”) to own shares in another company which in turn owns shares in “ACME” (directly or by owning shares in other companies). To find the names of all companies owned (directly or indirectly) by “ACME” we cannot use a
simple query using the close by operator as it would not function correctly upon encountering
the cycle. We can use the close unique by operator instead, as shown in the example below:

| E.8.10 | Find the names of all companies owned (directly or indirectly) by “ACME”.
| SBQL: | (Company where name = "ACME") close unique by hasSharesIn.Company |

A simple case of using the operator leaves unique by is shown below.

| E.8.11 | Calculate the approximation of the square root up to 5-th position after the dot.
| SBQL: | 1 as x leaves unique by ((integer)(100000 * (a/x + x)/2)/100000) as x |

In the above case we have no counter variable that terminates the closure. The termination is
achieved after the expression following leaves unique by returns the same value as previously.
However, the case is less safe, as it is possible in general that each next value of the closure
will always be different than the previous one (the final results may form a cycle).

As shown above, some advanced tasks may lead to very complex queries which semantics
could be difficult to grasp for the programmers. However, the complexity is caused by the
tasks rather than by the language. According to our tests, such tasks written e.g. in C++ may
take several source code pages. SBQL offers also other recursive querying options which may
support easier problem decomposition, in particular fixed point equation and recursive
procedures.

Sometimes, apparently quite easy tasks cannot be formulated without a transitive closure. In
example E.8.12 we show the case when calculating a proper bag of numbers require such an
operator.

| E.8.12 | Assume Emp objects with an attribute salary. For each interval \([i, i+100), i = 0, 100, 200, \ldots\) get the number of employees having the salary within this interval. The maximal salary is unknown for the person asking the query.
| SBQL: | (0 as i close by ((i+100) where i <= max(Emp.salary)) as i) join (count(Emp where salary >= i and salary < i+100) as c) |

The result is a bag of structures \{ i(lower_bound), c nbr_of_emps \}. The transitive closure is
used to calculate the bag \{ i(0), i(100), i(200), \ldots i(maxsal) \}, where maxsal is the maximal
salary rounded to full hundreds. Note that the number of elements in this bag is initially
unknown, hence the use of the transitive closure. Even such a simple query is impossible to
express in SQL.

### 8.2 Fixed Point Equations

In mathematics a fixed point equation has the form \(x = f(x)\), where \(x\) is a variable or (a vector
of variables) of type \(T\), \(f\) is a function \(T \rightarrow T\). For instance, such a function can look like
\(x = (2/x + x)/2\). The solution of this equation is known as a least fixed point. For our example
function the solution is the square root of 2. Well-known context-free grammars can be
considered least fixed point equations. Recursive procedures can also be described as least
fixed point equations. The essence of such equations is that they precisely model recursive
processes and structures. Starting from some initial value \(x_0\) we calculate \(x_1 = f(x_0), x_2 = f(x_1),
x_3 = f(x_2), \ldots\) etc. till reaching the fixed point, i.e. when \(x_n = x_{n+1}\). Sometimes reaching the
fixed point requires infinitely many steps, but mathematical apparatus is well prepared for
such situations. It may also happen that the fixed point cannot be achieved in this way (hence the above procedure will result in an infinite loop). This means that function $f$ must be specifically constrained; these constraints also form a mathematical theory (unfortunately, inapplicable for our purposes).

The theory of fixed point equations has great meaning in mathematics (in particular, it is the foundation of a semantic specification method known as denotational semantics). In case of SBQL we have introduced fixed point equations as a facility for programmers to formulate recursive queries. For some tasks such equations could be more legible than transitive closures or recursive procedures and could better support decomposition of the task into subtasks with clear conceptual meaning.

Essentially, the evaluation mechanism of our fixed point equations is the same as the described above mathematical procedure. However, we must define some important syntactic and semantic details, such as “variables”, because SBQL binding rules for names are more semantically specific than the simple semantics of mathematical variables. The syntax of SBQL fixpoint equations is as follows:

$$\text{query ::= fixpoint} \left[ \left( \text{\'set_of_variables\'} \right) \right] \left[ \left( \text{\'set_of_rules\'} \right) \right]$$

$$\text{set_of_variables ::= \variable \ \{ , \ \ \ \text{variable} \} }$$

$$\text{set_of_rules ::= rule\{rule\} }$$

$$\text{rule ::= \variable \ :- \ \text{query \ ;} \ ;}$$

Let $\text{fixpoint} (x_1, x_2, ..., x_m) \ \{ x_1 :- q_1; x_2 :- q_2; ... x_m :- q_m; \}$ be a fixpoint system, where:

- $x_1, x_2, ..., x_m$ are names of variables in this equation system,
- $x_{i1}, x_{i2}, ..., x_{in}$ are returned variables, $\{ x_{i1}, x_{i2}, ..., x_{in} \}$ is a subset of $\{ x_1, x_2, ..., x_m \}$,
- $q_1, q_2, ..., q_m$ are SBQL queries with free variables $x_1, x_2, ..., x_m$.

The basic semantics of this construct is the following:

1. Variables $x_1, x_2, ..., x_m$ are initialized to empty bags. More precisely, a new section with binders $x_1(\text{bag}()), x_2(\text{bag}())..., x_m(\text{bag}())$ is pushed at top of ENVS; ($\text{bag}()$ stands for empty bag).
2. Queries $q_1, q_2, ..., q_m$ are evaluated. Their results are on the top of QRES.
3. Pop ENVS.
4. If the results of $q_1, q_2, ..., q_m$ are equal to the values of $x_1, x_2, ..., x_m$, respectively, then the fixpoint is reached, hence go to step 5. Otherwise assign the results of $q_1, q_2, ..., q_m$ to the values of $x_1, x_2, ..., x_m$. More precisely, a new section with binders $x_1(\text{result of } q_1), x_2(\text{result of } q_2), ..., x_m(\text{result of } q_m)$ is pushed at top of ENVS. Then remove results of $q_1, q_2, ..., q_m$ from QRES and go to step 2.
5. Results of queries that are not among $q_1, q_2, ..., q_m$ (corresponding to variables $x_{i1}, x_{i2}, ..., x_{in}$) are dropped. Remaining results are changed into binders $x_{i1}(\text{result of } q_{i1}), x_{i2}(\text{result of } q_{i2}), ..., x_{in}(\text{result of } q_{in})$ and left at the top of the QRES stack. Hence only variables $x_{i1}, x_{i2}, ..., x_{in}$ can be used in an outside query that consumes the results of the fixpoint operator. If $\text{set_of_variables}$ is absent (in our syntax it is optional), then all the variables $x_1, x_2, ..., x_m$ form the final result.

The symbol :- we use instead of = to distinguish the fact that we are dealing with equations having the above semantics rather than with regular equality predicates. As queries $q_1, q_2, ..., q_m$ can reference variables $x_1, x_2, ..., x_m$, fixpoint equations provides recursive capabilities. A fixpoint query should be used within some non-algebraic operator (or within a for each statement) which pushes the binders from top of QRES onto the top of ENVS.
Note that a transitive closure query \( q_1 \text{ close by } q_2 \) can be written as a fixed point equation \( x = q_1 \cup x \cdot q_2 \), hence the general fixed point equation \( x = f(x) \) offers the power of transitive closures. However, it is not sure if such general fixed point equations would be sufficiently usable and understandable by the programmers. Moreover, such complex equations could be difficult to optimize and could result in unpredictable infinite loops. It also easy to see that the power of the transitive closure operator is sufficient to express any fixpoint equations. The fixpoint system

\[
\text{fixpoint}(x_1, x_2, \ldots, x_m) \{ x_1 \leftarrow q_1; x_2 \leftarrow q_2; \ldots x_m \leftarrow q_m; \}
\]

is semantically equivalent to the following query with the leaves unique by operator:

\[
\text{struct}(\text{bag}() \text{ group as } x_1, \text{bag}() \text{ group as } x_2, \ldots, \text{bag()} \text{ group as } x_m)
\]

leaves unique by

\[
\text{struct}(q_1 \text{ group as } x_1, q_2 \text{ group as } x_2, \ldots, q_m \text{ group as } x_m).
\]

The equivalence stems from the fact that in both cases the evaluation procedure is the same. However, fixpoint equations offer much more compact and legible formulation of some recursive problems.

Below we present examples of SBQL fixpoint equations; they refer to the schema from Fig.8.1.

**E.8.13** All components of a part named “engine”.

**SBQL:**

\[
\text{fixpoint}(\text{engparts})\{
myPart :- Part \text{ where name="engine"};
\text{engparts}:- myPart \cup (\text{engparts.Component.leadsTo.Part});
\}
\]

The first rule returns a reference to a part named engine. The second rule involves recursion on the engparts variable. The operator union in this query is the union of bags. Indeed, it may happen that the fixpoint equation returns a bag rather than a set. This makes our construct different from Datalog, which assumes sets (relations) as results of calculations. To illustrate this situation consider the parts graph from Fig.8.3.

![Fig.8.3. Example of a graph of parts](image)

In Fig.8.3 arrows represent instances of the relationship Component. We can see that the part “bolt10x30” is a component of (at least) two sub-parts of the “engine” part. Hence, to be consistent, the reference to the bolt10x30 object should occur in the engparts variable (at least) two times. The necessity of such semantics will be more evident in next examples.
Fixpoint equations are regular SBQL queries, fully consistent with the entire SBQL semantics. As such may be used as parts of other SBQL queries, for instance:

E.8.14 All unique components of a part named “engine”.

| SBQL: | distinct (fixpoint (engparts) { myPart :- Part where name="engine"; engparts :- myPart union (engparts.Component.leadsTo.Part); }) |

E.8.15 From all components of a part named “engine” take parts named “carburetor” (two queries below are equivalent).

| SBQL: | (fixpoint (engparts) { mypart :- Part where name="engine"; engparts :- mypart union (engparts.Component.leadsTo.Part);}. engparts where name = “carburetor”) as engcarb |
| SBQL: | fixpoint (engcarb) { mypart :- Part where name="engine"; engparts :- mypart union (engparts.Component.leadsTo.Part); engcarb :- engparts where name = “carburetor”; } |

In the first query we use the fixpoint query within an outer query. In the second query the same effect is achieved by three rules.

A fixpoint construct may use some variables as a way to break down the problem into smaller, more manageable parts, for instance (c.f. Fig.8.2):

E.8.16 For each component of the engine get the number of identical elements that the engine consists of.

| SBQL: | fixpoint (final) { engine :- (Part where name=”engine”) as x, 1 as howMany; engineParts :- engine union (engineParts.Component.((leadsTo.Part) as x, (amount*howMany) as howMany); final :- ((distinct(engineParts.x) as y). (y,sum(engineParts where x=y).howMany)); } |

Only variable final is returned as the fixpoint. The other two variables are used to perform calculations, but never returned, as their final values are inessential to the user. Variable engine is used to find the top element of the hierarchy (the “engine” part), while engineParts is the variable in which the results of recursive calculations are stored. Variables final and engine are not defined recursively – their purpose is to make the query easier to read and create. Note that in this case the fact that variables store bags (and operator union is the union of bags) is essential. If duplicates are removed, the result would be warped. Assume that in Fig.8.3 bolt10x30 occurs once in carburetor and once in starter. Hence the structure struct{x(1bolt10x30), howMany(1)} occurs two times within the result of the variable engineParts: 1bolt10x30 is the identifier of the bolt10x30 object (finally, 2 such bolts are components of the engine, while removing duplicates will cause that the number of such bolts will be 1).

The same principle is used in the next example.
Get the total cost and mass of the engine.

**SBQL:**

```sql
fixpoint (cost, mass){
  engine :- (Part where name="engine") as x, 1 as howMany;
  engineParts :- engine union
    engineParts.Component.((leadsTo.Part) as x,
    (amount*howMany) as howMany);
  detailsMass :- sum((Detail)engineParts. (howMany*x.mass));
  detailsCost :- sum((Detail)engineParts. (howMany*x.cost));
  addedMass :- sum((Aggregate)engineParts.
    (howMany*x.assemblyMass));
  addedCost :- sum((Aggregate)engineParts.
    (howMany*x.assemblyCost));
  cost :- detailsCost + addedCost;
  mass :- detailsMass + addedMass;}
```

Queries utilizing fixpoint equations, unlike those utilizing transitive closure, are capable of evaluating more than one recursive problem in each recursion step, in a manner similar to the Datalog. It is believed that such capabilities will allow users to formulate complex and intelligent “business rules”. This topic may be an interesting area for further research, although most of the practical recursive problems authors are aware of can be solved using only a single recursion.

Similarly to transitive closure operator, fixpoint equations may be used to perform recursive calculations without referring to the database at all. For instance, the example below shows a fixpoint equations version similar to Example E.8.8:

**E.8.18** Calculate the cubic root of $a$, according to the fixpoint equation $x = \frac{(a/x^2 + x)}{2}$, starting from $x = 1$ and making 50 iterations

**SBQL:**

```sql
fixpoint(x){
  y :- (1 as r, 1 as c) union y.(((a/(r*r))+r)/2 as r, c+1 as c) where c <=50);
  x :- (y where c = 50).r;
}
```

Next example is analogous to E.8.9:

**E.8.19** Calculate 50 Fibonacci numbers. The output should contain arguments and results of the Fibonacci function.

**SBQL:**

```sql
fixpoint(Fib){
  init :- (0 as a, 0 as fib, 1 as n);
  f :- init union f.(((a+1) as a, n as fib, (fib+n) as n) where a < 50);
  Fib :- f.(a as arg, fib as F);
}
```

In the example variable $a$ is a counter (which is the argument of the function), $fib$ represents a Fibonacci number, $n$ represents a next Fibonacci number.

If the graph to be closed contains cycles then the transitive closure is unsafe (thus the special closure variant `close unique by`). This problem does not occur for systems of fixed point equations, as function `distinct` can be used as usual for removing duplicates. For instance (c.f. Fig.8.2):
### E.8.20
Find the names of all companies owned (directly or indirectly) by “ACME”.

**SBQL:**

```
fixpoint(comps) {
  acme :- Company where name = "ACME";
  comps :- distinct(acme union comps.hasSharesIn.Company);
}
```

Assume the following class definition (ODRA) for typical genealogical trees:

```java
class PersonClass {
  name: string;
  birthYear: integer;
  sex: enum[“m”, “f”];
  alive: boolean;
  parent: ref Person [0..2] reverse child;
  child: ref Person [0..*] reverse parent;
}
```

Person: PersonClass[0..*];

In `PersonClass` we have used bidirectional pointers parent/child that represent relationships in the genealogical tree. Then we declare a collection of objects named `Person`. This structure can be used to formulate interesting queries with the help of fixed point equations.

### E.8.21
Get names and sex for all living and younger cousins of “Doe”.

**SBQL:**

```
fixpoint(result) {
  myperson :- Person where name = "Doe";
  ancestors :- myperson union ancestors.parent.Person;
  allcousins :- ancestors union allcousins.child.Person;
  livingYoungerCousins :- allcousins where alive and birthYear > myperson.birthYear ;
  result :- livingYoungerCousins.(name, sex);
}
```

### E.8.22
Get all cousins of “Doe” of the same generation.

**SBQL:**

```
fixpoint(result) {
  myperson :- (Person where name = "Doe") as p, 1 as gen;
  ancestors :- myperson union (ancestors.((p.parent.Person) as p, (gen-1) as gen));
  allcousins :- ancestors union (allcousins.((p.child.Person) as p, (gen+1) as gen));
  result :- ((allcousins where gen = myperson.gen) minus myperson).p;
}
```

These examples much remind typical cases that can be found in the Datalog literature. Of course, we are not sure if all examples of Datalog programs can be formulated in this way. However, we do not expect that SBQL fixed point equations alone would have the full algorithmic power, as they are one of many SBQL constructs and features.
8.2.1 Strong Typing

For simplicity, in the presented constructs we skip typing issues. Because fixed point rules can be recursive, type inferences can be problematic (as initially the type of a variable on the right side of a rule is unknown). Probably this is not a very difficult problem to solve, however to avoid it each variable could be typed. This makes the syntax of the rules a bit more complex; however, the strong type checking would be more efficient.

8.2.2 Optimization

Evaluation of a fixpoint system may be expensive, especially when bags to be assigned to variables are very large. There are several opportunities for optimization. All optimization methods that are developed for SBQL can be used, in particular, indices, factoring out independent subqueries, removing dead subqueries, pushing selections before structure constructors (joins), caching, factoring out weakly dependent subqueries and pipelining.

There are also methods specific for fixed point equations. The method from Datalog is known as semi-naive evaluation. The main idea of this technique is to divide the fixpoint rules into groups, which may be evaluated independently from the other groups, utilizing only the final results of previous group evaluation. For instance, in E.8.21 we can evaluate myperson at the beginning, because this rule does not depend on other variables. Then, we can evaluate ancestors, as this variable depends only on myperson and ancestors. Then, we can evaluate allcousins, which depends on ancestors and allcousins. Finally, we evaluate result that depends on allcousins and myperson. In general, the method is easy to implement through constructing a dependency graph that for our example is shown in Fig.8.4. Calculations of a variable can start when all its predecessor variables are calculated.

Another optimization opportunity concerns rules of the form

\[ x :- c \text{ union } f(x) \]

where \( c \) is some already counted part, independent from \( x \). All recursive rules that we have introduced in our examples have this form. Under some assumptions (distributivity of \( f \) w.r.t. elements of its bag argument) such a recursion can be turned into the iteration:

\[ x_0 = c; x_1 = f(x_0); x_2 = f(x_1); x_3 = f(x_2); x_4 = f(x_3); \ldots \]

The iteration is continued till some next component returns the empty bag. Then,

\[ x = x_0 \text{ union } x_1 \text{ union } x_2 \text{ union } x_3 \text{ union } x_4 \text{ union } \ldots \]

Probably the magic sets technique, also known from Datalog, can be adapted to SBQL queries – further research in this direction is needed.

8.2.3 Convergence of Fixpoint Equations

Evaluation of fixed point equations can result in an infinite loop. That is, the fixed point is never reached. For instance (c.f. Fig.8.2):
### E.8.23

Find the names of all companies owned (directly or indirectly) by “ACME” (without removing duplicates).

#### SBQL:

<table>
<thead>
<tr>
<th><code>fixpoint( comps ) {</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>acme :- Company where name = ”ACME”</code>;</td>
</tr>
<tr>
<td><code>comps :- acme union comps.hasSharesIn.Company;</code></td>
</tr>
</tbody>
</table>

If objects `Company` form a cycle that can be accessed from the ACME object via the `hasSharesIn` relationship, then the evaluation process of `comps` will never be finished. This situation can be tested, as usual, but without any guarantee that a test will be reliable. For instance, it is possible that the above example is convergent for “ACME” as a parameter, but it leads to an infinite loop for another parameter.

In general, the convergence of fixpoint equations depends both on data structures to be processed and on query operators that are used in queries on right of the rules. For instance, the equation `x = 1 − x` starting from `x = 1` will produce an infinite loop `1, 0, 1, 0, 1, 0,….` (despite there is a fixpoint `x = 0.5`).

In our opinion, there is no sufficiently general theory or algorithm that could discover and prevent all such cases. In Datalog there is the `stratification` concept that prevents infinite loops in case when a negation operator is used. However, there are a lot of other operators that may cause infinite loops, for instance, arithmetic minus, divide, set-theoretic difference, function `sinus`, etc. Moreover, as shown in E.8.22, infinite loops can be caused also by cyclic data structures, which can be explicit (navigation via pointers) or implicit (a cycle due to some non-evident dependencies among data). Hence some “rules of thumb” are necessary to prevent such cases, in particular:

- Clear understanding of a problem to be solved by a system of fixpoint equations,
- Being suspicious to all implicit or explicit cyclic dependencies in data to be processed,
- Testing the equations for various cases.

Of course, these rules do not prevent all such bugs, hope that eliminate them in sufficiently reliable manner.

### 8.3 Recursive Procedures and Functions

As almost all popular programming languages, SBQL allows for declaring procedures (and methods), including recursive ones. Any procedure can be recursive due to the stack-based semantics of procedure calls. There is no special syntax separating recursive and non-recursive procedures. In principle, the number of levels of recursion is unlimited, however, there could be some physical and performance constraints. In contrast to languages such as Java, C++, C#, etc., SBQL is unique due to the following features:

- SBQL functional procedures (methods) can return a bulk output and can be called within SBQL queries. Hence, SBQL functional procedures can encapsulate queries.
- Parameters of SBQL procedures can be SBQL queries too. Parameters can be transmitted in the `call-by-value`, `call-by-reference` and `strict-call-by-value` modes.
- Output returned by SBQL functional procedures can contain references. Hence, the output from a procedure calls can be used for updating.

As follows from the previous features, SBQL functional procedures can work as virtual updatable views. However, for the well recognized view updating problem we consider procedures and views as separate notions. (SBQL updatable views are the subject of a separate chapter).
In this chapter we do not specify precisely all the syntactic and semantic details of SBQL procedures. This is the subject of a separate chapter. Below we show on examples how SBQL recursive procedures can be used to formulate recursive tasks.

The syntax for SBQL procedures is shown below.

**Procedure declaration**

A procedure declaration consists of a procedure name, a typed parameter list, a type of its output and a list of imperative SBQL statements.

\[
\text{procDeclaration ::= name}'('parameter_list')' ['returntype'] '{statement_list}'
\]

\[
\text{parameterList ::= parDeclaration ; parDeclaration}
\]

\[
\text{parDeclaration ::= name ':' type [cardinality]}
\]

If the cardinality is not specified, the default cardinality [1..1] is assumed.

**Procedure invocation**

\[
\text{procInvocation ::= name}'('actual_par_list')'\]

\[
\text{actual_par_list ::= parameter ; parameter}
\]

\[
\text{parameter ::= query}
\]

**Return statement**

\[
\text{statement ::= return [query]+';'}
\]

**Semantics:** When a procedure is called, the statements in the procedure are evaluated in the consecutive order. When the \textit{return} statement is reached, the result of the query being parameter of that statement is put on top of QRES. If no \textit{return} statement is reached before the end of procedure evaluation, the procedure returns nothing (thus it is not a functional procedure).

Statements in SBQL procedures use SBQL queries. SBQL includes several imperative operators (object creation, assignment, deletion, etc.) and control flow statements (including loops) such as \textit{if}, \textit{while}, \textit{for each}, etc.

A simplest recursive procedure consists of a single return statement, which depends on a procedure parameter either returning an empty collection or returning the result from invocation of the same procedure with different parameter value. A sample recursive procedure finding all components of specified parts, along with their amount required to make these parts, is shown below (see Fig.8.1):

**E.8.24** Find all components of specified parts, along with their amount required to make these parts.

**SBQL:**

\[
\text{type PCType is record } \{ p: PartType; c: integer; \} \\
\text{SubPartsPC( myPC: PCType[0..*]): PCType[0..*] \{} \\
\text{\quad return} \\
\text{\quad if exists myPC then bag}(\text{myPC}, \text{SubPartsPC (myPC.p.Component.(leadsTo.Part) as p, (c * amount) as c})) \\
\text{\quad else bag()})
\]

The procedure takes a bag of structures as the parameter (named \textit{myPC}). Each structure contains a reference to a part object named \textit{p} and the amount of parts of this type named \textit{c}. Note that a query being the parameter of this procedure can have unlimited complexity.
| E.8.25 | Find all the details necessary to make 125 engines and 150 gear boxes. |
| SBQL: | \[(\text{Detail}) \text{SubPartsPC} ( \text{bag} (\text{Part where name = "engine"} \text{as} p, 125 \text{as} c), (\text{Part where name = "gear box"} \text{as} p, 150 \text{as} c)))\] |
| E.8.26 | How many bolts 5X50 are necessary to make 125 engines and 150 gear boxes? |
| SBQL: | \[\text{sum}(\text{SubPartsPC} ( \text{bag} (\text{Part where name = "engine"} \text{as} p, 125 \text{as} c), (\text{Part where name = "gear box"} \text{as} p, 150 \text{as} c)) \text{where} p.\text{name} = \text{"bolt5X50"}.c)\] |

The advantage of recursive procedures is simplicity of problem decomposition. A recursive task can be easily distributed among several procedures (some of which may be reused in other tasks), and calculations within a single procedure can be performed in easy to comprehend steps, comparable to the simplicity of fixed point equations. A procedure calculating the cost and mass of a part illustrating this possibility is shown in the next example. The procedure utilizes the previously defined \text{SubPartsPC} procedure in order to perform the recursive processing and then performs calculations, utilizing local variables.

| E.8.27 | Get the total cost and mass of a part given as a parameter. |
| SBQL: | \[\text{PartCostMass}(\text{myPart: PartType}): \text{record} \{ p: \text{PartType}; \text{cost: real}; \text{mass: real}; \} \{ \text{allSubParts: PCType}[0..*]; \text{detailsMass: real; detailCost: real;}; \text{addedMass: real; addedCost: real;}; \text{create local allSubParts( SubPartsPC( \text{bag}(\text{myPart as} p; 1 \text{as} c)) ));}\text{detailMass} := \text{sum}(\text{Detail}) \text{allSubParts.(c * p.mass)};\text{detailCost} := \text{sum}(\text{Detail}) \text{allSubParts.(c * p.cost)};\text{addedMass} := \text{sum}(\text{Aggregate}) \text{allSubParts.(c * p.assemblyMass)};\text{addedCost} := \text{sum}(\text{Aggregate}) \text{allSubParts.(c * p.assemblyCost)};\text{return} (\text{myPart as} p, (\text{addedCost+detailsCost} \text{as} \text{cost,});(\text{addedMass+detailsMass} \text{as} \text{mass});)\} \] |

| E.8.28 | Get the cost for all parts having the mass higher than 100. |
| SBQL: | \[(\text{Part as} p). (\text{PartCostMass}(p \text{ where mass > 100})(p, \text{cost}))\] |

Recursive procedures in SBQL offer many advantages, when compared to stored procedures in relational DBMSs. Most of them are consequences of the fact that recursive procedures in SBQL are a natural extension of the SBA, working on the same principles and evaluated by the same evaluation engine – unlike relational systems, where stored procedures are an addition to the system, but are evaluated separately from SQL queries. The main advantages of SBQL recursive procedures are the following:

- SBQL queries are valid as expressions, procedure parameters, etc.;
- The type system is the same;
- There is no impedance mismatch.
SBQL procedures may be also used as a way to reuse queries or fixpoint equations. Instead of writing a stand-alone fixpoint equation for a single use, it is possible to write a procedure utilizing the fixpoint equation, while providing a way to parameterize it easily. A sample procedure doing this is shown below:

<table>
<thead>
<tr>
<th>E.8.29</th>
<th>Get all subparts of a part being a parameter.</th>
</tr>
</thead>
</table>
| SBQL:  | \[
| \text{subParts}(\text{myPart}: \text{PartType}): \text{PartType}[0..*] \{ \\
| \quad \text{return distinct}(\text{fixpoint}(\text{parts})\{ \\
| \quad \quad \text{parts} :: \text{myPart union} (\text{parts}.\text{Component}.\text{leadsTo}.\text{Part}); \})\}; \}
|        |                                               |

Recursive procedures can also substitute fixed point equations or transitive closures, for example:

<table>
<thead>
<tr>
<th>E.8.30</th>
<th>Calculate the square root of (a), according to the fixpoint equation (x = \frac{a}{x} + x), starting from (x = 1) and making 50 iterations.</th>
</tr>
</thead>
</table>
| SBQL:  | \[
| \text{Root}(a: \text{real}): \text{real} \{ \\
| \quad \text{return internalRoot}(a; 1; 50); \}
| \text{internalRoot}(b: \text{real}; x: \text{real}; c: \text{integer}): \text{real} \{ \\
| \quad \text{if } c = 0 \text{ then return } x; \\
| \quad \text{else return internalRoot}(b; (b/x + x)/2; c - 1); \}
|        |                                               |

Formulation of recursive definitions is equally simple:

<table>
<thead>
<tr>
<th>E.8.31</th>
<th>Calculate (n)-th Fibonacci number.</th>
</tr>
</thead>
</table>
| SBQL:  | \[
| \text{Fib}(n: \text{integer}): \text{integer} \{ \\
| \quad \text{return if } n = 0 \text{ or } n = 1 \text{ then } n \text{ else } \text{Fib}(n-2) + \text{Fib}(n-1); \}
|        |                                               |

Similarly, recursive procedures can be used for processing cyclic structures, for instance (c.f. Fig.8.2):

<table>
<thead>
<tr>
<th>E.8.32</th>
<th>Find the names of all companies owned (directly or indirectly) by “ACME”.</th>
</tr>
</thead>
</table>
| SBQL:  | \[
| \text{visited}: \text{string}[0..*]; //global variable storing names of visited objects \\
| \ldots \\
| \text{allOwnedBy}(\text{comp}: \text{CompType}): \text{CompType}[1..*] \{ \\
| \quad \text{delete visited;} \\
| \quad \text{intAllOwned}(\text{comp}); \\
| \quad \text{return Company where name in visited;} \\
| \} \\
| \text{intAllOwned}(\text{comp}: \text{CompType}) \{ \\
| \quad \text{if not comp.name in visited then} \{ \\
| \quad \quad \text{create visited( comp.name );} \\
| \quad \quad \text{for each (comp.\text{hasSharesIn}.\text{Company} as own do} \\
| \quad \quad \quad \text{intAllOwned ( own );} \\
| \quad \} \\
| \}
| SBQL:  | \text{allOwnedBy(Company where name = ”ACME”)}                        |

In the example we use a global variable \text{visited} to avoid starting next navigation via \text{hasSharesIn} from an already visited object. This task can be of course solved in many different ways.

In this chapter we have presented and compared several implemented and postulated techniques for expressing recursive problems in database query and programming languages.
We have considered three techniques: transitive closures (four variants), fixed point equations and recursive procedures and functions. All of them are implemented in the ODRA system; hence we have gained some experience. Practical examples have shown that usability of these techniques can be different in various situations and none of them is dominating or disadvantageous. Although recursive tasks are not much frequent, in some important situations presence of the techniques in a database programming language can much simplify the programmers’ effort.
9 Storing and Processing Irregular Data (Semi-Structured)

Irregular data, that is, data that do not follow some fixed format, are properties of many current technologies, especially based on XML. Irregular data are also specific to conceptual modeling, which frequently requires some forms of loose formats. Irregularity in data has many forms and presents a lot of issues, especially concerning strong type checking and query languages. Irregular data is the reason of a lot of naïve research, inconsistent solutions, poor imagination of the developers and false opinions. In this chapter we try to discuss more important issues related to irregular data and present our views and concepts concerning irregular data in the context of SBA and SBQL and object-oriented store and query models.

The most recognized concept related to irregular data is a *null value* known from the relational model, relational databases and SQL. The concept, inevitable and obvious in relational data structures, becomes quite difficult in query and programming languages. In SQL it is supported apparently with little care about semantic consistency and consequence. Depending on SQL constructs, nulls are treated very differently and all the design seems to be contradictory to common logic and sense. However, there is perhaps no fault of SQL designers. The devil of chaotic design is inside the null value concept itself. According to [Date86b, Date86c, Date92b, Subi01b, Subi96, Subi98] there is no consistent way to introduce nulls to database structures and corresponding query and programming languages.

A new wave known under the term “semi-structured data” [Abit97, Abit97c, Abid97d, Bune95, Bune96, Bune97, Lahi99, Quas95, Semi97, ] approaches the problem from a different angle and actually does not refer to null values. The problem arises in the context of more loose and weakly typed data structures that are common for some applications, in particular, for Web and XML technologies. However, the semi-structured data problem inherits a lot from the problem of null values.

In general, any DBMS must eventually deal with complete computational and pragmatic power of programming interfaces. Hence, features related to irregular data must be combined with all aspects of database systems: data description, query languages, updating and other imperative constructs, typing, object-orientedness, procedures, methods, views, active rules, metadata, etc. Support for irregular data is especially challenging if one assumes that the data is to be processed by strongly typed programming languages, such as C++, C# and Java. Usually such languages support regular data formats only. The necessity of mapping between irregular data and regular formats must lead to some impedance mismatch. Hence, the support for irregular data must be designed with an extreme care concerning conceptual simplicity, minimality, consistency, universality and openness for external data processing tools.

During the development of SBA and SBQL we address the problem of irregular data in object bases and consider how they are to be manipulated in the integrated query/programming languages addressing object models. The difference between our approach to the problem and the approaches known in the literature is that we extend the scope of issues related to irregular data to all the aspects that are necessary to make a professional object-oriented database query and programming language. Our approach to irregular data is very simple and is based on the idea of collections and cardinalities of UML. Null values are represented as absent data. Any data that can be absent are to be typed by cardinalities with lower value equal to zero, for instance [0..1] or [0..*]. In particular, optional data are treated as a collection being empty or having exactly one element. To process such a data the programmer can uses facilities that are specific for processing collections rather than for processing values. In this way we can deal
with nulls without introducing an explicit a null value. Within this approach all the disadvantages of null values presented in [Date86b, Date86c, Date92b] become irrelevant. This chapter discusses the situations that are related to this new attitude to nulls and presents corresponding SBQL solutions.

9.1 Null Values

Many authors (more than 500 papers), see e.g. [Dubo88, Dyre95, Gess90, Gess91, Imie91, Lenz91, Leve91, Libk94, Liu 90, Liu 90b, Piro92, Roth89, Roth91, Sadr91, Tana89, Zani84, Zica90], have associated null values with “incomplete information”, making a lot of research aiming at solving this apparently important problem. After dozens of years almost all this research has been commonly recognized as scholastic, with little (or perhaps no) practical impact in the domain of database management systems. Some of these researches, however, are claimed to be applicable in other domains, in particular, artificial intelligence, data mining and data warehouses.

From the practical point of view, null values are loosely related to the problem of uncertain or missing information. The term irregular data better expresses the problem. It concerns situations when particular information does not fit exactly the predefined data format, or when some spots in the storage media (or in their formal or conceptual model) are not filled in by meaningful data. The interpretation of the fact that a particular data spot is unfilled is a data semantics issue, outside of any formal model.

Actually, SQL adopts such attitude to null values as described above. The meaning of SQL null values is not predefined. Null values present some technical feature in programming that can be used for different purposes and informally interpreted by designers, programmers, database administrators and other users. Because of many sources of null values and many reasons for which they are introduced it is difficult to imagine that another, more semantically specific approach to null values makes a sense. As an example, if we assume that almost all persons in some collection are not handicapped, then the value of the attribute handicapKind can be null for almost all objects. Only handicapped people have some information within this attribute. But interpretation of these nulls belongs to informal business semantics of data. In this case nulls have nothing in common with uncertain or incomplete information – just nulls are used as regular and certain information determining that corresponding persons are not handicapped. Some methodologists might not recommend using nulls in such a way, but actually this would be only recommendation and the decision anyway belongs to application and database designers.

The irregularity of data is relative to the typing system that is assumed in the given database and its query/programming environment. Irregularity with respect to some older data models (such as the relational one) is no more irregularity if the typing system is prepared for it and there are corresponding constructs that serve all the types in a corresponding query and programming language. Hence the true attitude to irregular data is to change them into regular ones through preparing a proper typing system. Such data are to be addressed by a database query and programming language with all the universality. However, irregular features require extra programming options, thus resulting languages can be more (or even too much) sophisticated for programmers.

In the development of SBQL we tried to introduce irregularities to object data structures without compromising the simplicity of the language. Eventually, however, any design must be acknowledged by common practice of language’s users.
9.2 Variants

In the domain of PLs there is another well-known feature that can be considered as irregular data. It is called “variant” in the Pascal family of languages, or “union” in the C family. In the following we will use the term “variant” (“union” is already used as an SBQL query operator). Originally (e.g. in Pascal) variants were introduced to save the memory space. This motivation no more holds for current data models where variants have mostly conceptual meaning. A variant may be required when objects of a conceptually similar type can change the collection of their attributes. For example, an Employee object may concern regular employees and junior employees. In a variant for a regular employee the object contains the attribute salary and no attribute scholarship, while in a variant for a junior employee – vice versa.

Variants and null values are conceptually similar. If a null value we interpret as an absent attribute, than the assumption that an attribute A can be null-valued might be modeled as a variant of record types R(A, B, C,...) and R(B, C, ...). The notions of null values and unions are, however, not equivalent. For example, if some record type involves n attributes that can be null valued, then the number of corresponding variants is $2^n$. Variants cover also the situation when names of attributes are the same, but types are different. It is interesting to note that the PL community noticed variants and ignored null values, and the database community did v/v. Because variants do not match the concept of a relation, they were unacceptable for the proponents the relational database model. This is one of many cases when a stiff mathematical frame is an impediment for the progress.

One can observe that in object-oriented models variants are unnecessary, as they can be substituted by specialized classes. However, classes mostly bear conceptual information from the business model. It would be improper to burden the model by some technical reasons such as variants. Variants would cause the appearance of many artificially introduced classes, with exotic names and unclear meaning. Moreover, if there is more than one variant in one business object type, then this may result in the combinatorial explosion of (permuted) specialized classes. If a type introduces n binary variants on different attributes, then the number of specialized classes that conceptually cover this situation is $2^n$. This presents an argument that variants should be introduced as a feature orthogonal to classes.

Variants, as introduced in Pascal, require a special attribute (so called discrimination attribute) that allow during runtime determining which variant is currently processed. Such an attribute makes it possible strong type checking of variants (which in this case must be delegated to runtime). In C variants (unions) were introduced without the discrimination attributes; hence the responsibility for determining a variant is in hands of the programmer. To this end he or she can choose any option and any information. The solution makes strong typing of variants impossible.

In the literature there is a subdivision between exclusive and non-exclusive variants. Exclusive variants mean that in the actual object we can choose A or B, but not A and B together. For instance, an Employee record can have the attribute salary or the attribute scholarship, but not the both. Non-exclusive variants do not imply such a constraint and hence are easy to be substituted by optional data: non-exclusive variant between A and B with cardinalities [0..1]. Hence, only exclusive variants present essentially new situation for conceptual modeling, typing and programming capabilities.

Variants are also similar to dynamic object roles, as described in the AS2 object store model. Dynamic object roles provide capabilities much more general and flexible than the capabilities assumed in variants. The major difference concerns conceptual modeling.
Variants can be introduced for any technical reason, while dynamic object roles are assumed to have some business-oriented semantics. In particular, each dynamic object role must possess a distinguished external name, while this is not required for a variant. Actually, however, these distinctions between variants and dynamic object roles are secondary from the technical point of view. Object roles can be provided as a substitute and generalization of variants. This leads to some new situations with strong typing. Such an approach to typing roles is described in the PhD thesis by A. Jodłowski [Jodl03].

9.3 Typing of Null Values and Variants

Irregular data require a new attitude to typing, especially to static (compile time) strong type checking. Some authors argue that due to the irregular nature of data strong typing is impossible: such data are type-less or schema-less. We disagree with such opinions. Without a typed data schema the programming is awkward. In particular, a database schema is a carrier of business-oriented data semantics and determines the structure and representation of data. It is difficult to imagine that the programmer can write programs without any schema, because in such cases he or she must guess the structures from examples of data or apply a generic programming where everything, including data names, is perceived as strings. At least for conceptual modeling, types or schemas must be introduced anyway, explicitly or implicitly. This is the motive for such languages as DTD or XML Schema. The question arises how such facilities can be used for static type checking, for determining representation of data, for resolving some ambiguous situations (such as ellipses and automatic coercions), for query optimizations, etc. Within SBQL we have proposed a typing system that we call “semi-strong”, to underline the fact that our types are prepared to typing irregular data, with all the features that are usually associated with types, such as conceptual modeling, static type checking of queries and programs, resolving ambiguities in queries and programs and preparing information for query optimization.

Concerning null values and variants, static type checking (during compilation time) has limits because both a null value and a variant are runtime features. Hence in such cases type checking must be usually delegated to runtime. Alternatively, developers of a language can assume that null values and variants are outside the typing system and instead of typing errors they assume exceptions in reaction of some illegal runtime situation (e.g. when a null is an argument of an arithmetic operator). Such exceptions, however, require anyway semantically clean detection of null values and variants during runtime.

More information on typing irregular data will be given in a section devoted to typing.

9.4 Current Proposals Concerning Irregular Data

The current view on null values is materialized in relational databases and SQL. A null value can be stored in a database and returned by an SQL expression if it cannot be evaluated correctly. This may happen because of null-valued arguments of an operation, as well as in the case of wrong arguments of operations. For example, function sum returns NULL for an empty argument table. SQL does not allow explicit comparisons of nulls and ordinary values but involves a special predicate is [not] null. A special function if_null returns an attribute’s value if it is not null, or some constant value otherwise. Comparison of an expression which returns NULL with any other value returns a third truth value UNKNOWN; in the WHERE clause, however, it is equivalent to FALSE. In embedded SQL the admission of null values requires for every such attribute two variables in an underlying host language program. An
extra indicator variable is used to store boolean information determining if the particular value of an attribute is null or not.

Several authors (notably Date and Darwen) point out difficulties related to null values, which make the semantics of user/programmer interfaces obscure and inconsistent. Date [Date86, Date92] presents a severe criticism of null values in the relational model (“...the null value concept is far more trouble than it is worth”, “…the SQL null value concept introduces far more problems than it solves”). He gives many striking examples of anomalies implied by null values. For instance, although in SQL “officially” every null value is distinct (thus A = B returns UNKNOWN if both A and B return nulls), the group by and unique operators treat them as identical, and aggregate functions sum, avg, min, max, count totally ignore them. Another striking example: if relation R contains numerical attributes A and B which could be null valued, than in general select sum(A+B) from R may return the result different from select sum(A) + sum(B) from R. Date concludes that these flaws are not only the property of an inadequate SQL design, but they are the inevitable consequence of the idea of null values.

The descendant model of SQL also promotes the use of null values. The SQL 99 standard [ANSI94, Melt93, Melt99] proposes to introduce an arbitrary number of application-specific null values such as “Unknown”, “Missing”, “Not Applicable”, “Pending”, etc. It is not clear, however, how this feature will be reflected in the semantics of particular query/manipulation constructs.

Variants are proposed in IDL of the OMG CORBA standard [OMG02, OMG95], with an explicit discrimination attribute. It is assumed that they are to be addressed in programming languages. Only C and C++ provide corresponding facilities. A similar concept is proposed in the ODMG standard. However, the standard contains no construct to deal with variants in the query language OQL and contains no suggestion how variants are to be treated by the assumed strong typing system. As far as we know, actually there is no powerful query language consistently dealing with variants. Null values are introduced in the ODMG standard in a manner that is much more restrictive than it is done in SQL. This is perhaps motivated by the possible problems with nulls that the designers of the ODMG standard want to avoid. No operators such as outer joins are allowed in ODMG OQL. In general, object-oriented database models neglect the problem of irregular data. Perhaps SBA and SBQL are first proposals that address powerful object-oriented models and support non-trivial forms of irregular data.

There are also proposals to make loosely typed or typeless data structures. Examples are XML and RDF-oriented technologies. For instance, in XML one can introduce any tags or attributes within tags and to put inside them any values\textsuperscript{15}. In RDF one can change ad hoc the type of some entity. Such approaches are even advocated as “more flexible”, hence having advantages over disciplined data structures. However, the flexibility should not be confused with chaos. Tags (data names) and types are features of a schema, which is necessary for the user to ask correct queries. If tags and types are unknown, then the user or programmer must guess them and to discover the meaning of values basing on some side information, which can also be poorly defined. Our conclusion is that loosely typed or typeless data structures are unmanageable, especially for very large databases. Such proposals can be used for small, non-responsible applications, where the user can see the entire database (and its next possible states) before asking queries. In real applications such situations are difficult to imagine.

\textsuperscript{15} This can be of course constrained by DTD or XMLSchema, but such constraints are not obligatory.
9.5 Irregular Data in Theories

The fundamental defect of theories w.r.t. null values and other forms of irregular data is the *scope mismatch*. The phenomenon is similar to the impedance mismatch and concerns the difference between the scope of theories and the scope in which irregular data need to be considered in practice. Theories usually address an idealized query language with very limited capabilities (e.g., SPJ queries in the relational algebra, or the language of some predicate logic). The scope for null values is much wider and should include:

- All advanced query constructs, such as grouping, aggregate functions, quantifiers, ordering;
- Imperative constructs based on queries: creating, updating, inserting, deleting;
- The interface for embedding a query language into a programming language;
- Database semantic enhancements: views, database procedures, integrity constraints, deductive rules, active rules;
- Object-oriented extensions: ADTs, types, classes, interfaces, encapsulation, inheritance;
- Static strong or semi-strong type checking;
- Privacy, security, transactions;
- Interfaces for end users and programming interfaces: graphical query languages, 4GLs, and other.
- Metadata (an internal representation of a database schema)

The scope mismatch is the main reason for a very low applicability of the theories addressing null values and other forms of irregular data. From the position of designers and vendors of DBMS it is very risky to introduce some particular solution without a clear view how it could be expanded to the whole conceivable environment of database application programming. In our opinion, theories concerning null values in databases have severely impeded the progress rather than supported it.

9.6 Irregular Data in Object Databases

In practice there are many sources of irregular data. Sometimes they can be qualified as uncertain information, but in general irregularities are related to specific aspects of data storing and processing.

- Information is irrelevant, for example, “nickname” for a person who has no nickname.
- Information is known, but not filled yet because data input lasts some time. For example, a new employee is hired, all his/her data are already introduced to the database, except “salary” which has to be input by another administration division.
- Information is valid only for fixed time; after it must be nullified (to prevent errors) and then input again or recalculated; for example, weather or stock market forecasts.
- Information is known and could be filled in, but it is considered inessential for the particular domain of applications. For example, in some airline ticket reservation system the nationality of passengers is not filled in for domestic flights.
- Information consists of “legacy” records, with limited amount of information, and actual records having an extended set of attributes. Hence in the integrated resources legacy records must possess null values.
- The result of some processing is an intermediary data structure containing null-valued elements; e.g. outer joins.
Nowadays, there are next reasons to deal with irregular data. The following aspects of object database technologies may require them:

- **Object-oriented analysis and design.** In many cases the admission of null values, unions and repeating data presents an important facility for the analyst or designer. For example, the designer can stick two classes into one introducing some null-valued attributes; and vice versa.

- **Schema evolution.** As a result of schema evolution some new attributes may appear, some may disappear, some may change type, and some single attributes can be changed into repeating ones. This naturally leads to variants and optional data.

- **Interoperability.** Heterogeneous and/or distributed databases require resolution of missing or conflicting data values that occur when semantically identical data items may have some attribute values different of missing in some data sources.

- **Web and XML.** HTML and XML files are weakly formatted. There is a lot of valuable research devoted to querying semi-structured data, especially XML; however, there is little doubt that they will require efficient programming techniques to deal with their irregularities and inconsistencies.

- **Data warehouses.** There is a tendency for collecting information concerning particular topics (e.g. a stock market) from various heterogeneous sources. Analytical processing of such heterogeneous information (statistical analysis, overview reports, data mining, etc.) requires capabilities of integrated query/programming languages that will make querying null values and variants possible.

Notice, however, that null values and variants are additional programming options that can be easily avoided both on the stage of database design and on the stage of programming. For instance, if attribute A of type integer can be null valued, it is enough to introduce two attributes: attribute A with no null allowed and an additional Boolean attribute is_A_null?, with value true indicating that the value of A should not be taken into account because it is invalid. This approach is actually used in host languages embedding SQL. It corresponds to discrimination attribute that is used to distinguish variants. The attribute is_A_null can also store more information about the status of A, for example, “not initialized”, “pending”, “irrelevant”, etc. Such an approach does not require introducing irregularities of data explicitly; however, it could be more difficult for conceptual modeling and makes description of objects more complex.

### 9.7 Date’s Default Values

As an alternative to null values C.J.Date and other authors propose the concept of default values, with the intention that default values work as nulls. They are ordinary values that are filled in into a tuple in the case of absent information. Default values can be determined in a relational schema. According to Date, default values do not require any special options concerning irregular data in databases and in programming languages.

Unfortunately, this claim is obviously false and based on fundamental misunderstanding. Null values and other irregularities are conceptual issues, while default values present purely technical option to record irregularities. If the programmer forgets to consider default values as a special mean for storing irregularities, then probably his/her program will be formally correct, but having conceptual flaws from the point of view of the business that is supported by the program. To avoid this danger, default values must be indicated in the conceptual database schema, which would become more complicated. Processing default values may require special care or special options. For instance, if for salaries the default value would be -
1, then calculation of the average salary through the aggregate function \( \text{avg} \) would be erroneous. The programmer cannot forget that before calculating the average he/she must filter all objects with \( \text{salary} = -1 \). Moreover, for some domains there could be no good default, e.g. for the Boolean type. Default values are also extremely error-prone: if the programmer forgets or not properly recognizes defaults, then the typing system will not be able to recognize an error and it appears during runtime with no warning.

Note that this intention of default values is unusual for programmers. Usually a programmer expects that default values are the most common values that are filled in during object creation, for example the value “no” for the attribute \( \text{IsSmoking}? \) Using default values with another intention will require – at least – another term to denote them and special explanation in manual.

Summing up, the proposal of using default values as the substitution for null values is doubtful and risky in the same way as the criticized null values are. In our opinion, default values are not worth to consider seriously as a special option in data structures and query languages.

### 9.8 SBA - Approach to Irregular Data

We assume that a schema is an inevitable property of any data store, including storing irregular data. A schema is necessary for conceptual modeling. The programmer before writing a query or a program must clearly understand what the database contains and how it is logically organized. Such a schema can be informal or even not explicit, but anyway must exist.

If a schema is expressed formally, it can be used for dynamic (runtime) checking of processed objects. For example, DTD and XMLSchema play this part for XML repositories. A bit more formality and discipline is necessary if we assume that types declared within a schema are to be used for static (compile time) type checking of queries and programs.

The new qualities introduced by irregular data to this assumption are the following:

- A schema should be able to express the fact that some data are optional and may not be present in a particular data instance. The feature is specified by the cardinality \([0..1]\) and is equivalent to the SQL clause “NULL IS ALLOWED”. In this way every data that can be optional is modeled as a bag which is empty or has exactly one element. All operators that address bags can be applied to such a bag, in particular, quantifiers. Properly designed implicit type coercions between bags and individual elements (and v/v) make it possible to reduce the conceptual overhead related to optional data to minimum.
- A schema should be able to express the fact that some data are alternative. This issue is known as “exclusive variants”. We assume that a variant must possess a discriminator that allow for type checking of variants during runtime.
- A schema should be able to express the fact that some data can be repeated some number of times, including zero times or any number of times. The issue is known as cardinality of collections. Cardinalities can also specify pointer links (association instances) among objects.
- The schema language should allow orthogonal combination of the above features for any data hierarchy level.
- Any binding to a data that is absent results in empty bag rather than in a special value “null”. Although this looks as a very subtle change, there are essential consequences. We show that the idea makes it possible to overcome difficulties encountered in SQL and can
be smoothly combined with variants, and object-oriented concepts. It can also be consistently incorporated in query languages and programming languages integrated with queries.

As far as we are aware of, this model is the first, simple, intuitive and universal idea, which is capable to express uniformly all features related to semi-structured information and is acceptable to an average database engineer. The idea is consistently incorporated into data structures, a query language, programming interfaces and other capabilities of the prototype object-oriented database management system ODRA. The ODRA query and programming language SBQL is much more powerful than Lorel and UnQL, well recognized for their capabilities to process irregular data. Moreover, SBQL has very precise, intuitive semantics and is seamlessly integrated with programming constructs, programming abstractions and object oriented notions.

### 9.9 Querying Optional Data, Variants and Repeating Data

Absent data (including variants) imply the necessity of special care in queries. Consider the SBQL schema (from the ODRA system):

<table>
<thead>
<tr>
<th>E.9.1</th>
<th>Database schema involving departments and their employees</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SBQL:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>class EmpClass {</td>
</tr>
<tr>
<td></td>
<td>instance Emp: {</td>
</tr>
<tr>
<td></td>
<td>name: string;</td>
</tr>
<tr>
<td></td>
<td>address : AddressType [0..1]; //absent data allowed</td>
</tr>
<tr>
<td></td>
<td>salary: real[0..1]; //absent data allowed</td>
</tr>
<tr>
<td></td>
<td>job: string;</td>
</tr>
<tr>
<td></td>
<td>worksIn: ref DeptClass[0..1] reverse employs; //absent data allowed</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>class DeptClass {</td>
<td></td>
</tr>
<tr>
<td>instance Dept: {</td>
<td></td>
</tr>
<tr>
<td>employs: ref EmpClass [0..*] reverse worksIn;</td>
<td></td>
</tr>
<tr>
<td>boss: ref EmpClass[0..1]; //pending absent data allowed</td>
<td></td>
</tr>
<tr>
<td>dname: string;</td>
<td></td>
</tr>
<tr>
<td>budget: real;</td>
<td></td>
</tr>
<tr>
<td>location: string [1..*]; //multiple values allowed</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>Emp: EmpClass [0..*];</td>
<td></td>
</tr>
<tr>
<td>Dept: DeptClass [0..*];</td>
<td></td>
</tr>
</tbody>
</table>

The schema declares one type AddressType and two classes EmpClass and DeptClass. Then it declares two collections of objects, Emp and Dept, of the defined classes. For Emp objects the attribute address is optional. Similarly, in the Emp object the attribute salary is optional too. In Dept objects the reference employs is multi-valued, the reference boss is optional and the attribute location can assume one or more values.

Consider the query **Emp where salary >1000**. If for some employee the attribute salary is absent, then binding name salary will return an empty bag, which is compared by > with
1000. The comparison > is not defined for such a case. We can therefore consider the following possible definitions of the semantics:

- Predefined truth value: in this case the formula will return FALSE.
- The formula will return the third logical value UNKNOWN.
- A runtime type error or exception, i.e., the case is semantically forbidden.

The first approach leads to unreliable programs, because, e.g., some obvious tautologies do not hold. For instance, not (salary > 1000) is not equivalent to salary <= 1000. The second case requires introducing a lot of features to the integrated query/programming languages, which would process the UNKNOWN value. As experience has shown, such a new truth value is extremely cumbersome in constructing and using the language. Although two previous cases are allowed in some languages (e.g. SQL, LINQ and OCL), our decision is restrictive: only the last case is acceptable. We do not accept some false “user friendliness” in which queries are simpler at the cost of clean semantics and reliability of applications.

The standard query capabilities that exist in SBQL can be adopted to handle the last case. Depending on preferences, the designer or programmer can choose between the options which we present below. For each described capability we list two options. In the first one, marked by (a), an identifier of an Emp object with absent salary is included in the result of the query. In the second option, marked by (b), an identifier of an Emp object with absent salary is excluded from the result.

<table>
<thead>
<tr>
<th>E.9.2</th>
<th>Get employees earning more than 1000</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SBQL:</strong></td>
<td>// Quantifiers:</td>
</tr>
<tr>
<td>a)</td>
<td>Emp where forall salary as s (s &gt; 1000)</td>
</tr>
<tr>
<td>b)</td>
<td>Emp where forsome salary as s (s &gt; 1000)</td>
</tr>
<tr>
<td>a)</td>
<td>// The SBQL function exists tests salary:</td>
</tr>
<tr>
<td>b)</td>
<td>Emp where if exists(salary) then salary &gt;1000 else true</td>
</tr>
<tr>
<td></td>
<td>(Emp where exists(salary)) where salary &gt; 1000</td>
</tr>
<tr>
<td>a)</td>
<td>// The SBQL function count calculates the size of the argument:</td>
</tr>
<tr>
<td>b)</td>
<td>Emp where if count(salary) = 1 then salary &gt;1000 else true</td>
</tr>
<tr>
<td></td>
<td>(Emp where count(salary) = 1) where salary &gt; 1000</td>
</tr>
<tr>
<td></td>
<td>//In some languages operators and, or are “lazy”, i.e. if the</td>
</tr>
<tr>
<td></td>
<td>// argument before or returns true (argument before and returns false)</td>
</tr>
<tr>
<td></td>
<td>// then the second argument is not calculated.</td>
</tr>
<tr>
<td></td>
<td>// With such semantics we have one more case:</td>
</tr>
<tr>
<td></td>
<td>Emp where not exists(salary) or salary &gt;1000</td>
</tr>
<tr>
<td></td>
<td>Emp where exists(salary) and salary &gt; 1000</td>
</tr>
<tr>
<td></td>
<td>// In general lazy operators can lead to problems in query languages,</td>
</tr>
<tr>
<td></td>
<td>// because some query optimization method based on rewriting</td>
</tr>
<tr>
<td></td>
<td>// can change the order of arguments of boolean operators.</td>
</tr>
</tbody>
</table>

The optional data may concern not only atomic objects, but also complex ones. Note that a null value concerning a complex attribute is not expressible in relational systems. SBQL can also deal with this case:
E.9.3  Get names and cities of programmers; if \textit{address} is absent, return the string “no address”.

\textbf{SBQL:} 
\begin{verbatim}
(Emp where job = "programmer").
(name, if exists(address) then address.city else “no address” )
\end{verbatim}

Similar methods concern variants. Querying variants does not imply any special option in SBQL; however variants with discriminators require special features of a strong typing system.

SBQL is also prepared to process any other cardinality of collections. The query below processes \textit{location} of \textit{Dept}, which has the cardinality \([1..*]\).

E.9.4  Get cities hosting all departments.

\textbf{SBQL:} 
\begin{verbatim}
(unique(deref(Dept.location)) as deptcity)
where forall Dept (deptcity in location)
\end{verbatim}

In the first line the query collects locations of all departments, and then dereference converts \textit{location} identifiers to strings. Then, \texttt{unique} removes duplicate strings. Each element of the result is named \textit{deptcity}. The second query line selects cities hosting all departments.

As a side effect of the above assumptions, aggregate functions have the same semantics as in SQL: absent data do not influence the result. Consider the query

E.9.5  Get the average salary of all employees.

\textbf{SBQL:} 
\begin{verbatim}
avg(Emp.salary)
\end{verbatim}

For \textit{Emp} objects with no \textit{salary} sub-object the name \textit{salary} occurring in the query will return the empty table. This table will be merged with other tables according to the union operator; hence it does not influence the result. This approach is simple, consistent and should be easily understood by the programmers. Consider the Date's example

E.9.6  SQL:
\begin{verbatim}
select sum(A+B) from R
select sum(A) + sum(B) from R
\end{verbatim}

where the first query may return a result different from the second query. In SBQL the first query cannot be formulated as \texttt{sum(R.(A+B))} because if \textit{A} or \textit{B} denote an absent attribute then the operator \texttt{+} fails. The correct formulation is:

E.9.7  \textbf{SBQL:} 
\begin{verbatim}
sum(R.(A as a, B as b).(a+b))
\end{verbatim}

The operator ‘,” (structure constructor) returns an empty table if any of the arguments is absent. Hence, \texttt{a+b} is never evaluated with \texttt{a} or \texttt{b} returning an empty table. The second query in SBQL

E.9.8  \textbf{SBQL:} 
\begin{verbatim}
sum( R.A ) + sum( R.B )
\end{verbatim}

can hardly be confused with the first one.
In the following we present more queries with pitfalls related to incompleted data.

<table>
<thead>
<tr>
<th>E.9.9</th>
<th>Get name, salary and department name for employees earning less than 2222</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>$(\text{Emp where } \text{salary} &lt; 2222 \land (\text{name}, \text{salary}, \text{worksIn.Dept.name})$)</td>
</tr>
<tr>
<td></td>
<td>(Possible dynamic type error if some employee has no salary)</td>
</tr>
<tr>
<td></td>
<td>Safe version: $\text{Emp where forsome (salary as } s \land (s &lt; 2222)) \land (\text{name}, (\text{if exists(salary)} \land \text{deref(salary)} \land \text{else 0.0}), \text{worksIn.Dept.name})$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E.9.10</th>
<th>For each department get its name and the sum of salaries of employees being not bosses.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>$((((\text{Dept as } d) \land \text{join ((sum(d.employs.Emp.salary) - (d.boss.Emp.salary)) as } s)) \land (d.dname, s))$</td>
</tr>
<tr>
<td></td>
<td>(Possibly dynamic type error if some Dept has no employees or if all employees in some Dept have no salary or if some boss have no salary)</td>
</tr>
<tr>
<td></td>
<td>Safe version: $((((\text{Dept as } d) \land \text{join ((d.employs.Emp) subtract (d.boss.Emp)) as } s)) \land \text{groupas dEmps}) \land (dEmps.d.dname, (\text{if exists(dEmps.s.salary)} \land \text{sum(dEmps.s.salary)} \land \text{else 0.0})$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E.9.11</th>
<th>Is it true that each department employs an employee earning the same as his/her boss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>forall (\text{Dept as } d) \land forsome ((d.employs.Emp subtract d.boss.Emp) as e) \land (e.salary = d.boss.Emp.salary)</td>
</tr>
<tr>
<td></td>
<td>(Possibly dynamic type error if some employee has no salary)</td>
</tr>
<tr>
<td></td>
<td>Safe version: forall (\text{Dept as } d) \land forsome ((d.employs.Emp subtract d.boss.Emp) as e) \land forsome (e.salary as es) \land forsome (d.boss.Emp.salary as bs) \land (es = bs)</td>
</tr>
<tr>
<td></td>
<td>Note that if a boss has no salary, then the corresponding department is not taken into account.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E.9.12</th>
<th>For each employee get the message containing his/her name and the percent of the annual budget of his/her department that is consumed by his/her monthly salary.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBQL:</td>
<td>$\text{Emp.(&quot;Employee &quot; + name + &quot; consumes &quot; + (salary * 12 * 100 / (worksIn.Dept.budget)) + &quot;% of the &quot; + worksIn.Dept.dname + &quot; department budget.&quot;);}$</td>
</tr>
</tbody>
</table>
(Possibly dynamic type error if some employee has no salary)

SBQL: Safe version:

\[(Emp \text{ join (if exists}(salary) \text{ then deref}(salary) \text{ else } 0.0) \text{ as } s).\]

"Employee " + name + " consumes " + (s* 12 * 100/worksIn.Dept.budget) + "% of the " + worksIn.Dept.dname + " department budget."

For each interval \(<n,n+999>, n = 0, 1000, 2000, 3000, ... \) get the message (string) containing the number of employees having the salary within this interval and the interval itself. The output messages should have proper grammatical forms (suffixes -s for nouns (n) and verbs (v)).

SBQL:

\[(0 \text{ as } i \text{ closeby } (i+1000 \text{ where } i \leq \text{ max}(Emp.salary)) \text{ as } i)\]

\[\text{join } (\text{count}(Emp \text{ where salary } \geq i \text{ and salary } < i+1000) \text{ as } c)\]

\[\text{join (if } c=1 \text{ then } ("n as n, "s as v) \text{ else } ("s as n, "s as v)).\]

(c+" employee"+n+" earn"+v+" between "+i+" and "+(i+999) as message)

E.9.13

For each interval \(<n,n+999>, n = 0, 1000, 2000, 3000, ... \) get the message (string) containing the number of employees having the salary within this interval and the interval itself. The output messages should have proper grammatical forms (suffixes -s for nouns (n) and verbs (v)).

(Possibly dynamic type error if some employee has no salary)

SBQL:

\[(0 \text{ as } i \text{ closeby } (i+1000 \text{ where } i \leq \text{ max}(Emp.salary)) \text{ as } i)\]

\[\text{join } (\text{count}(Emp \text{ where exists}(salary)) \text{ where salary } \geq i \text{ and salary } < i+1000) \text{ as } c)\]

\[\text{join (if } c=1 \text{ then } ("n as n, "s as v) \text{ else } ("s as n, "s as v)).\]

(c+" employee"+n+" earn"+v+" between "+i+" and "+(i+999) as message)

9.10 Capabilities Equivalent to Outer Joins

In the relational model the outer join is an important operator that is relevant for semantic modeling and design methodologies, and convenient for programmers. In many cases the ordinary (inner) join is not satisfactory because it loses information. For example, the name and the department name of an employee can be obtained by the query (providing dno column within Emp and Dept tables):

E.9.14

SQL:

\[\text{select name, dname from Emp, Dept where Emp.dno = Dept.dno}\]

The resulting table, however, will not contain information on such employees for which the dno attribute is null valued. The outer join enables the user to formulate the query returning a tuple \(<\text{name, NULL}>\) for an employee having null valued dno attribute. The operator is usually considered non-primitive, as it can be expressed by other operators of the relational algebra. Date [Date92c] advocates the “default style” outer joins which avoids null values. There could be more specialized outer joins: left-hand side outer join ignores nulls that occur at right side join argument table, and right-hand side outer join does vice versa.

A similar operator could be also useful in object-oriented query languages. Consider the corresponding query in SBQL:

E.9.15
It returns no information on employees for which the `worksIn` sub-object is absent. We can avoid special options because the existing SBQL operators are sufficiently powerful. For example, assuming that the user would like to return the string “???” for `Emp` objects with absent `worksIn`, the above query can be formulated as:

| E.9.16   | SBQL: | `Emp.(name, if exists(worksIn) then (worksIn.Dept.dname) else “???”)` |

There are other solutions, still without introducing special features to SBQL. For instance, we can use `groupas` operator:

| E.9.17   | SBQL: | `Emp.(name as n, (worksIn.Dept.dname) groupas d)` |

This solution is just in the spirit of SQL: instead of null, however, we return an empty bag named `d`. Such a solution can be applied in all cases that in relational languages require outer joins and can be specialized to cases equivalent to left-side and right-side outer joins.

### 9.11 Default Values in SBA

A class object in an object-oriented PL may store several kinds of invariants inherited by class members. Default values can be considered as such invariants. We distinguish two roles for them:

- Initial values for some attributes, which are used when a new object is created. They are physically copied into a created object, thus need not to be inherited. Such default values are used only by a distinguished method commonly denoted as `new`. This kind of default values is not relevant to irregular data.
- Common default values, which are dynamically inherited by class members and can be overridden by values within a particular member. Such a value is used in situations when a given member object has a corresponding attribute absent. Default values stored inside classes have the same external name as a corresponding attribute. They can be atomic or complex.
Fig. 9.1. Overriding of default values

Default values in the object-oriented approach do not imply special features in query languages, assuming the model and scoping/binding rules as explained in previous chapters. However, we must consider a new semantics of assignments. When one updates a value which actually is inherited from a class, the semantics should not update the value, but create a new sub-object with the value determined by the right hand side of the assignment. For example, in the case shown in Fig. 9.1 the assignment

\[
\text{E.9.18} \quad \text{SBQL:} \quad \text{for each } \text{Supplier where } \text{Sname} = "\text{Black}" \text{ do } \text{Status} := 45;
\]

should result in creating a new atomic object \(<\text{i}_{\text{new}}, \text{Status}, 45>\) and then inserting it into the Black's object. Updating of class properties via its members should not be allowed.

Note that the runtime structures should allow the navigation from the identifier of a default object to the identifier of the object that is currently processed (to make within it a proper change). In SBA such navigation is quite easy, because the environment of the currently processed object is currently on the top of the stack. However, some minor modification to the stack-based mechanism could be necessary.

The presented definitions of default values capabilities can be easily generalized, for example, in order to cater for cases when an inherited property can be not only a "static" object, but also a functional procedure. Assume that when the status of a supplier is not defined, it is calculated as the number of products she supplies. In this case instead of introducing the object \(<..., \text{Status}, -1>\) to the \text{SupplierClass} we introduce the following functional procedure (a method), where \(SP(\text{Sno}, \text{Pno},...)\) denotes the relation connecting suppliers and parts:

\[
\text{E.9.19} \quad \text{SBQL:} \quad \text{integer procedure } \text{Status} \{
\quad \text{return count}(\text{SP as } x \text{ where } x.\text{Sno} = \text{self}.\text{Sno});
\};
\]
There are many examples of situations in the conceptual database design, when such dynamic defaults might be useful. As far as we know they have not been considered in the database literature.

Note that the procedure, like all inherited procedures, is evaluated within an environment of a Supplier object, thus the second occurrence of Sno in the procedure's body is properly bound to Sno of Supplier (what is indicated by self). Semantically, the construct SP as x creates so-called binders, i.e., named runtime entities. For detailed semantics related to binders see chapter devoted to defining auxiliary names.

9.12 Default Values and Scoping/Binding Rules

Default values stored within classes require special attention concerning scoping/binding rules. In some situations it is difficult to determine consistent scoping rules that obey natural expectations of the programmer. The expectations can be summarized in two points:

First visit the internal properties of an object; if there is no proper attribute, and then look for the default within its class, superclass, etc.

During execution of a method first visit its internal environment, then visit its class, superclass, etc. looking for local properties; next, visit the internal properties of an object being the receiver of a message.

These rules are contradictory for default values. For example, assume that a method m is stored within a class c1 and refers to an attribute a stored within object O; O is an instance of a class c2, and c2 contains a sub-object a storing a default value for the attribute a, Fig.9.2.
Assume that class \( C_1 \) inherits from \( C_2 \), the object \( O \) received the message \( m \) being a property of \( C_2 \), and within the body of \( m \) there is a reference to the attribute \( a \). The natural order of visiting particular environments during the binding name \( a \) occurring within the body of \( m \) is shown on Fig.9.2a as a sequence of curved arrows:

- Visit the local environment of \( m \);
- Visit private and exported properties of \( C_2 \) (to bind some other methods, class variables or procedures that may be called within \( m \));
- Visit the sub-objects of \( O \);
- Visit exported properties of the classes that \( O \) belongs to; in this case, the exported properties of \( C_2 \). The default \( a \) will be bound here if \( a \) was not bound in the previous step;
- Visit the rest of the environment (database objects, global temporary objects, functions and procedures from available global libraries).

The questions arises what will happen if \( C_1 \) and \( C_2 \) are the same class? In such a case the above rules make our idea of default values invalid for the method \( m \), because the default \( a \) will be visited and bound before the attribute \( a \), Fig.9.2b.

There are of course several ways to avoid this conflict. For instance, inside a method if a default attribute is to be bound, it checked if an attribute with the same name exists in the
currently processed object. The above case shows, however, that for object-oriented database programming languages the scoping/binding rules may imply some pitfalls, hence all possible situations should be carefully analyzed, designed and tested.

9.13 Possibility of False Binding

If a query language is not strongly typed, irregular data together with assumed full orthogonality of query operators may cause a binding problem. To explain it, consider the query:

| E.9.20 | Get employees who have salary and earn the same as Brown |
| SBQL: | $(Emp$ where $exists(salary))$
$\quad$ where $salary = ((Emp$ where $name = "Brown")$.salary$)$ |

If the Brown’s salary is absent the query should cause a runtime error. However, during the binding of the third occurrence of $salary$ the environment stack is as shown in Fig.9.3.

Since the top of the environment stack contains no $salary$ binder the search will be continued in lower sections of the stack. The section below contains the $salary$ binder, thus the binding will succeed. As a result, the predicate after the second $where$ will be $TRUE$ for any $Emp$ having a $salary$. The query will return all such employees.

Obviously, the result is wrong. Let us analyze the reason. The third $salary$ occurrence is referring to the Brown’s object; if evaluated independently, the nested sub-query ($(Emp$ where $name = "Brown")$.salary$)$ will return the empty bag. However, because the query is nested, $salary$ is considered as referring to the outer query. In terms of our stack model, the search for the $salary$ binder should be finished at the top of the stack, but because of absent data there is no such a binder and the search is continued in lower sections of the stack. Hence such a false binding is caused by the lack of information about the semantic qualification of the second occurrence of the name $salary$ in the query. This information should be somewhere given, and should influence the scoping rules and/or structures introduced in the model.

This information is usually present in the type of an object. It can be used during compilation to determine statically which section of the environment stack (relatively to its top) is relevant for binding a particular name; so the problem does not occur. Analogously, the problem can be easily solved by introducing to the $EmpClass$ a special entity named $salary$, which the only role is to stop the search in lower sections of the stack; the binding of name $salary$ to such objects will return an empty table.
If the data environment is untyped, and there is no possibility to introduce the corresponding information to a class, the programmer must be supported with special facilities to deal with such cases. Among various options we present the following. We assume that every name occurring in a query can be augmented by an arbitrary number of names of its attributes. Each attribute name is preceded by a keyword with or without. For example,

\textit{Emp with name with salary without job}

This phrase is evaluated (bound) as a single semantic unit (i.e., name, salary and job are not bound independently). It returns the references to all Emp objects which are present in the actual scope and satisfy the “with” and “without” conditions: that is, contain the attributes name and salary but do not contain the attribute job. Inherited properties are not taken into account.

The described facility gives the programmer a full control over the binding, despite possibly defaults and absent data. For example, assuming salary and job can be absent, and a default job is stored within the EmpClass, the following query

\begin{verbatim}
E.9.21
SBQL: (Emp without salary with job) where job <> "clerk"
\end{verbatim}

returns references to all employees containing a job subobject, have no salary and are not clerks. Note that the default jobs and salaries do not influence the result. The query E.9.20 can be formulated as follows:

\begin{verbatim}
E.9.22
Get employees who have salary and earn the same as Brown
SBQL: (Emp with salary) where salary = ((Emp with salary where name = "Brown").salary)
\end{verbatim}

9.14 Assignment to Absent Object

If an object is typed by the cardinality [0..1] in general the simple assignment may result in an exception if the object does not exist. Hence each assignment to such an object should be preceded by testing whether the object exists and then inserting it, for instance:

\begin{verbatim}
E.9.23
Let Brown earn 5000.
SBQL: for each (Emp where name = "Brown") as B do {
    if exists B.salary then B.salary := 5000
    else B <<= 5000 as salary;
}
\end{verbatim}

The operator <<= means “create and insert” – on the right side there is a query that creates an object and on the left side there is a reference to an altered object. If there are many objects that are typed in this way, such assignments can be annoying and the program will become much less legible. A solution of this problem is to introduce a special operator (or to extend the meaning of the := operator) that creates an object if it does not exist, for instance:

\begin{verbatim}
\textbf{We strongly discourage making untyped programming interfaces for queries. SBQL is strongly typed. As will be shown in next sections, strong typing has a crucial role for query optimization. One can argue that discovering typing errors during compilation time is not so important and we can live without it, c.f. SQL. However, lack of query optimization totally undermines the query interface for its users.}
\end{verbatim}
Let Brown earns 5000.

SBQL: (Emp where name = "Brown").salary := 5000;

Unfortunately, such a solution will not work if we compile this statement as usually and try to do the proper operation at runtime. The left side of the assignment will return an empty bag and it will be impossible to recognize what is the name of an object to be inserted and where it should be inserted. Recognition of the situation during compilation and corresponding altering of the generated code seem too challenging (if not hopeless). Because the object to be assigned can be the result of a query with any complexity, which may involve function, method and view calls, in the general case altering of the generated code could be totally impossible.

Such a feature requires altering the run time structures, on the same principle as default values discussed above. Assume that each declaration of an object (attribute, sub-attribute, etc.) with lower cardinality 0 always means that the corresponding runtime class will be augmented by a dummy object having the same name as the object and with some dummy value. For instance, for EmpClass from E.9.1 after compilation we insert dummy objects

\(<id\_dummy\_1, address, dummy>, <id\_dummy\_2, salary, dummy>, <id\_dummy\_3, worksIn, dummy>\),

where dummy is a value distinguished internally that cannot be used by the programmer and cannot be stored in the object store. Its only role is binding of names that cannot be bound in normal ways. Having such a dummy object named \(x\) it will be bound in all situations where \(x\) cannot be bound in normal way, because it is absent. After such a binding the system can recognize that updating is to be performed on the dummy object, and instead of that, it goes to the top of the ENVS stack, which in this case should contain the environment of the currently processed object. Hence, it can recognize to which object the new object is to be inserted. For instance, for example E.9.24, if Brown has no salary, then the query returns id\_dummy\_2. An attempt to make the assignment will cause navigation to the top of ENVS, finding the reference of the Brown object, and inserting into it the sub-object \(<\text{some}\_\text{new}\_\text{id, salary, 5000}>\). To make this possible the system should be a bit lazy with reducing ENVS. It should not be reduced before the operator := is completed.

Note that the dummy value is not equivalent to the null value. A dereference of a reference to a dummy object should return an empty bag. Dummy objects and the dummy value are never used by the programmer and he or she needs not to be aware of such a feature. It is assumed to be an internal option of the binding mechanism that is necessary to solve the problem of assignments to absent objects.

9.15 Typing Irregular Data in SBQL

Well-known typing systems are not prepared to type irregular data. Actually, only the type system assumed in the XMLSchema is able to specify some non-trivial irregularities in data. However, XMLSchema is used as a grammar of XML files rather than a typing system for strong type checking of queries or programs. In general, because of new database models, new semantic properties of query languages and semi-structured data, all the known strong typing systems developed for programming languages are too limited and not flexible enough. Sometimes there are even opinions that irregularity in data (semi-structured data) is contradictory to strong type checking of queries and programs.

We disagree with such opinions. Irregularity of data is relative. If a typing system is prepared to deal with irregular data, they are no more irregular and become ordinary and typical.
Actually, the problem is in developing a new typing system, because for such situations the known typing systems are not prepared to this goal. This concerns also advanced typing systems based on inclusion and parametric polymorphisms, F-bounded polymorphism, etc.

We have developed a new strong typing theory and implement it for SBQL. We call it “semi-strong” just because it deals with semi-structured data as described above. We distinguish internal and external type systems. The internal type system reflects the behavior of the type checking mechanism, while the external type system is used by the programmer. A static strong type checking mechanism simulates run-time computations during compile time by reflecting the run-time semantics with the precision that is available at the compile time. Roles of the SBQL typing system are the following: compile-time type checking of query operators, imperative constructs, procedures, functions, methods, views and modules; user-friendly, context dependent reporting on type errors; resolving ambiguities with automatic type coercions, ellipses, dereferences, literals and binding irregular data structures; shifting type checks to run-time, if it is impossible to do them during compile time; restoring a type checking process after a type error, to discover more than one type error in one run; preparing information for query optimization by properly decorating a query syntax tree.

The internal SBQL type system includes three basic data structures that are compile-time counterparts of run time structures: a metabase, a static environment stack and a static result stack. Static stacks process type signatures – typing counterparts of corresponding run time entities. Signatures are additionally associated with attributes, such as mutability, cardinality, collection kind, type name, multimedia, etc. For each query/program operator a decision table is provided, which determines allowed combinations of signatures and attributes, the resulting signature and its attributes, and additional actions.

Processing types with cardinalities require delegating type checking to run time. For each creating, inserting and deleting operation performed on an object typed with non-trivial cardinalities the SBQL compiler inserts a piece of code that checks the corresponding cardinality during run time.

More about semi-strong typing system of SBQL is the subject of the special chapter and in description of the ODRA system. An example of SBQL types is presented in E.9.1.

9.16 Irregular Queries

Irregular data are usually associated with irregular queries that are impossible to ask in SQL. Examples of irregular queries can be as follows:

- Get names (as strings) of objects having an attribute storing the string “Monthy Pyton”.
- Get names (as strings) of attributes that store the string “Monthy Pyton”.
- Get objects of any type that store the string “Monthy Pyton” at any attribute and at any level of the hierarchy of their attributes.
- Get the number of all attributes in the person object with the name “Tom Jones”.
- Get Emp objects with an attribute having the name starting with letter “S” and with the value terminated by the string “logy”.

We can give more complicated examples of such unusual queries. All such queries require generic capabilities, sometimes beyond the assumed strong typing system. It is difficult to say if such queries are necessary. If such a necessity exists, it usually means that the design of a data schema is ill. However, current experiences with XML and its query language XQuery show that for some environments such queries make a sense.
Actually, the above queries require some generic functions that remain reflexive capabilities. We list some of these functions:

- Navigation from a reference of an object to references of all its components (toSubordinates). Obviously, the typing system is violated; hence it must be somehow changed.
- Navigation from a subobject to its direct superobject (toOwner).
- Returning the collection of names of all objects (allObjects).
- Given a reference to an object, returning its name as a string (nameOf).
- Given a string, use it as a data name (asName).
- Navigation from a reference to an object to its class/type (toClass, toType).

We can invent of course more such functions and perhaps their number is unlimited. It may happen that some conceivable query is still not possible within the given set of such functions, but inventing them is quite easy (although not sure if makes a sense).

Within this set of functions augmenting SBQL we can ask the following queries:

| E.9.25 | Get names (as strings) of objects having an attribute storing the string “Monthy Pyton”.
| SBQL: | $((allObjects as m join toSubordinates(m) as s) where s = “Monthy Pyton”).nameOf(m)$ |

| E.9.26 | Get Emp objects with an attribute having the name starting with letter “S” and with the value terminated by the string “logy”
| SBQL: | $((Emp as m join toSubordinates(m) as s) where nameOf(s)[1] = “S” and s like “*logy”) |

The examples has shown that such (relatively simple to implement) functions are able to rise the power of a query language. However, it is not such that such a power would not be the reason of some non-disciplined design of a database schema.

### 9.17 In Closing …

Although it is a common belief that null values and other forms of semi-structured information are one of the inevitable features of database applications the object-oriented database research and technology tend to neglect the issue. Practical proposals, such as the ODMG standard, present very few examples of relevant capabilities. Theories proposed for object bases and their query languages, such as object algebras, F-logic, comprehensions and structural recursion, do not supply comprehensive solutions. Moreover, the discussed SQL approach ultimately leads to numerous flaws and inconsistencies.

In this chapter we have presented a systematic approach to the problem based on the idea of cardinalities assigned to types of objects and briefly described our solutions. These can be summarized into the following recommendations:

We agree with Ch. Date: a special generic value called null (nil) should be absolutely avoided. We argue that it acts like a little devil that is able to spoil the semantic clarity and consistency of all language constructs; this especially concerns query languages.
Ch. Date suggests using default values instead of nulls. In general, we doubt if this approach offers advantages and reduces the possibility of inconsistencies and programmers’ errors. However, in this chapter we present simple methods to incorporate default values into classes.

In all cases when the use of default values is inconvenient or impossible (e.g., unions, repetitions, complex attributes) binding a name of an absent object should result in an empty collection, which can be processed by standard language facilities, such as exists, count, and quantifiers.

Static strong type checking of queries and programs addressing semi-structured data in the context of object-oriented query languages is not considered in currently typing systems and their theories (including polymorphic ones). SBA and SBQL offer semi-strong typing system based on compile-time simulation of run-time semantics. The approach makes it possible to provide the type checking as far as it is possible for irregular data.
10 Appendix 1. Principles of Query/Programming Languages

Majority of the principles presented below are speculative ideals that stem from aesthetic feelings and considerations. We have to strive to ideals but in real life it is impossible to achieve all of them. Moreover, ideals can be contradictory to real-life critical factors such as performance, implementation effort and development time. Many principles are violated due to the bottom-up development, where initially designers think about a small and easy language, but after success of the language they are trying to extend it incrementally with all the functionalities that are required by different applications and groups of users. For instance, we can claim that strong type checking is a principle of programming languages, but if some language was initially developed without this feature it would be extremely difficult to introduce it incrementally without affecting thousands of already existing programs and violating knowledge, habits and customs of thousands of programmers. The bottom-up development syndrome affects many commercial artifacts, including SQL, object-relational databases, PHP, Web Services, and others. In such artifacts we can observe a lot of violated principles, thus they are commonly perceived as non-aesthetic and sometimes even chaotic. Nevertheless, they are useful, and this property should be considered as a meta-principle dominating over all other principles. The usability of a language is not in contradiction to the mentioned below principles; indeed, we believe that eventually they support the usability.

- **Orthogonality**: keep unrelated features unrelated.
- **Compositionality**: avoid big syntactic and semantic patterns.
- **Separation of concerns** (Dijkstra): conceptually independent concerns (aspects) have to be independent program or data modules.
- **Universality**: make sure the language covers the assumed domain.
- **Generality**: use a language feature for many purposes.
- **Parsimony** (the Occam’s razor): avoid redundant features.
- **Naturality**: employ innate human psychology and the ways of programmers’ thinking.
- **Simplicity**: make the language easy to learn, use, document, maintain, implement and extend.
- **Clean, formal semantics**: not necessarily mathematical, but strong.
- **Clean, formal model of data structures to be queried and processed**: not necessarily mathematical, but strong.
- **Don’t neglect extreme cases**: empty strings, empty sets, empty loops, null-values, etc.
- **Genericity**: parameterized program abstractions, higher-order functions, reflection, and extensible syntax.
- **Extensibility**: make sure the language can grow.
- **Openness**: let the programmer to use external systems and specialized tools.
- **Avoiding semantic anomalies**: no exceptional features and irregular treatment.
• **No semantic reefs**: human (programmer) understanding and machine processing coincide.

• **Abstractions**: procedures, functions, views, types, ADTs, classes, methods, modules, encapsulation, etc.

• **Correspondence**: the methods of binding are the same for declarations and parameters.

• **Conceptual closure**: introducing a feature A enforces introducing features B, C, D, ..., Z.

• **Programming safety**: static program type checking, dynamic object type checking, assertions, constraints.

• **Semantic relativity**: identical properties of parent and nested constructs.

• **Conceptual continuation**: a bigger task can be smoothly composed from smaller ones that are already done and encapsulated.

### 10.1 Orthogonality

Every combination of language features that makes sense should be allowed (providing the given combination does not violate typing constraints). Orthogonality allows the designers to minimize language definition and to increase its retrieval or computational power. Orthogonality supports language implementation because shorter language definition implies shorter implementation code and easier optimization. Orthogonal languages are much easier to learn and use. Orthogonality may become essential in the case when programs in the given language are generated automatically as an output of some tools, e.g. CASE tools or graphical user interfaces.

### 10.2 Compositionality

Avoiding big syntactic and semantic patterns (cf. the SQL `select...from...where...group by...having...order by...pattern`), avoiding implicit semantic dependencies between distant pieces of the code, avoiding tangled structure of programs with no clean modules and their hierarchy (cf. the `goto` statements). Compositionality means that query/program pieces that have their own semantic meaning should be as small as possible (for example, only literals and names) and can be orthogonally composed into bigger and bigger units by a set of operators, where each of them has a small number of arguments (ideally, one or two arguments).

### 10.3 Correspondence and Conceptual Closure

The use of names and bindings in programming languages must be consistent. In particular, the mechanism of bindings names used in declarations of data structures must coincide with the mechanism of binding parameters of procedural abstractions; the methods of binding are the same for declarations and parameters. This one-to-one correspondence includes methods of parameter passing (call by value, call by reference etc.) and time of binding (static or dynamic).

Each new feature A of a query/programming language should be smoothly combined with all the already existing features B, C, D,..., Z. The new feature A must be smoothly integrated with the typing system of the language, its binding mechanisms, scoping rules, imperative
constructs, programming abstractions, database abstractions, etc. It must be consistently introduced into documentation, user manuals and training courses. For instance, if the designers of a language decide to introduce a new data structure, say, dynamic arrays, the corresponding questions are: “How such an array will be typed?”, “Can it be an element of a structure?”, “Can it be an element of a collection?”, “Can contain collections as elements?”, “Can be a parameter of a procedure?”, “Can be returned as a procedure output?”, etc. Because a new feature have to be combined with all the existing features, one can conclude that the complexity of semantics and pragmatics grows in square to the number of features introduced in a given query/programming language. Thus implementation and optimization efforts grow probably more than in square to the number of features. Hence advices:

- Avoid redundant features (the Occam’s razor);
- Make the power through orthogonality;
- Reduce kinds of introduced data structures to some minimal but still universal set;
- Generalize existing features rather than introduce incrementally new and new features independently.

10.4 Substitutability

If a subclass \( B \) inherits from a superclass \( A \), then an object \( b \) being an instantiation of \( B \) can be used in all places of a query/program in which object \( a \) being an instantiation of \( A \) can be used. For instance, an object \( \text{Student} \) can be used in all places when the object \( \text{Person} \) can be used, because a class \( \text{StudentClass} \) inherits from the class \( \text{PersonClass} \). Although this principle (known also as LSP, Liskov Substitutability Principle) seems to be obvious, it must be taken with care, at least for three reasons. The first concerns updating, where straightforward application of this principle leads to anomalies or (at least) different semantic interpretations. The second case concerns parameter passing, where the requirement of strong type checking leads to a long (and a bit academic) dilemma concerning co-variance and contra-variance. The last, most difficult case concerns the case when an object name is a property of its type (this does not occur in programming languages, but is a strong rule in database schemata). For such a case the object-oriented model based on substitutability makes little sense and must be superseded by some more general model. Substitutability, together with the open-close principle, is also contradictory to the straightforward concept of collection, fundamental for databases. From these three concepts you can take any two, but not the three together.

10.5 Open-Close

Each class can be closed for modifications, but it is still open for specializations. The principle is the basis for program reuse and method polymorphism. For instance, we can buy within some compiled library a class \( \text{Person} \), together with all the methods that are implemented for the class. We have no possibility to change anything in this class, because its source code is unavailable. Nevertheless, we can make specializations of this class by classes \( \text{Student}, \text{Employee}, \text{Customer}, \) etc. Each specialization inherits the implementation that is already done within the class \( \text{Person} \) and augments the implementation by new methods, specific for the given specialization. Moreover, each specialization can override some method implemented within the class \( \text{Person} \) by a specialized method implemented within this specialized class and having the same name (plus compatible parameters) as the name of the method within \( \text{Person} \).
The open-close principle must be taken with care, because it is contradictory to the substitutability principle and the straightforward concept of collections of objects, fundamental for databases. From these three concepts you can take any two, but not the three together.

10.6 Semantic Relativism of Objects

If some object \( O_1 \) can be defined, then object \( O_2 \) having \( O_1 \) as a component can also be defined. There should be no limitations concerning the number of hierarchy levels of objects. Objects on any hierarchy level should be treated uniformly. In particular, an atomic object (having no sub-objects inside) should be allowed as a regular data structure independent from other structures. The relativism of data structures implies the relativism of corresponding query capabilities, i.e. there should be no difference in language concepts and constructs acting on different object hierarchy levels. Traditionally, an object consists of attributes; an attribute consists of sub-attributes, etc. Assuming the semantic relativism of objects there is no need for such distinction: attributes, sub-attributes, pointer links between objects, procedures, methods, views, etc. are objects too. The semantic relativism of objects radically cuts the size of database model, the size of specification of query languages addressing the model, the size of implementation, and the size of documentation. It also supports easier learning of both a database model and a corresponding query language. By minimizing the number of concepts the semantic relativism of objects supports development of a universal theory of query languages, which is necessary to reason about query optimization methods.

10.7 Total Internal Object Identification

Each object which could be separately retrieved, updated, inserted, deleted, authorized, indexed, protected, locked, etc. should possess a unique internal identifier. The identifier is not printable and the programmer never uses it explicitly. A unique internal identifier should be assigned not only to objects on the top level of their hierarchy, but to all sub-objects, including atomic ones. If some atomic objects create a repeating group, e.g. a person has many hobbies, each object in the group should possess a unique identifier. For persistent objects (i.e. database objects) their identifiers should be persistent too, i.e. invariant during all the life of the objects.

We are not interested in the structure and meaning of internal identifiers, e.g., they can be disc addresses, RAM addresses, some symbolic internal names, some tuple identifiers (tid-s), pairs \(<\text{tid}, \text{attribute name}>\), triples \(<\text{relation name}, \text{primary key value}, \text{attribute name}>\), etc. For us it is essential that all objects and all their sub-objects can be unambiguously identified through its internal unique name. The principle makes it possible to make references and pointers to all possible objects, thus to avoid conceptual problems with binding, scoping, updating, deleting, parameter passing, and other functionalities that require references as query primitives.

The total object internal identification principle is not satisfied in value-oriented database models, such as the relational model, models based on the mathematical logic (Datalog), functional database models, etc. This, however, introduces a lot of limitations and inconsistencies. For instance, without assuming that each tuple has a unique identifier it is impossible to give the correct semantics of the SQL \textit{delete} clause or SQL cursors. Even more, without identifiers of values that are stored within a tuple it is impossible to express the semantics of the SQL \textit{update} clauses. Value-based database models are thus idealistic and
unrealistic. More advanced functionalities (e.g. updating) sooner or later will require correcting them by introducing some concept of internal identifiers.

Unfortunately, the principle is also not satisfied by XML. However, some functionalities developed for this technology require internal identification of XML sub-objects, thus special methods and languages have appeared to repair this flaw, c.f. Xpath, Xlink and Xpointer. All such inventions would be unnecessary assuming the total internal identification (by symbolic addresses) for parsed XML objects. In this case all the functionalities assumed in Xpath, Xlink, Xpointer, XQuery (and perhaps, next and next XML querying tools) can be covered by a single, simple and much more powerful query language such as SBQL.

10.8 Orthogonal Persistence

Traditionally, persistent data was stored in databases and they were collections. On the other hand, popular programming languages refer to volatile data only (i.e. programming variables) and such data are individual. (Some languages, such as Pascal and Java, introduce volatile collections too, but with severe constraints on their construction and use.) Nowadays programming languages become more and more abstract, thus the border between different storage kinds is more and more transparent. Moreover, some professionals anticipate that in few years databases will be stored in main memory only; there are already such prototypes and commercial solutions. The size of main memories is reaching dozens of gigabytes, i.e. is comparable to the capacity of disks, but the access time to main memory is millions time faster. In some solutions (e.g. a CORBA-based middleware) the programmer is in fact unaware if he/she deals with persistent or volatile data. Lack of individual data stored in databases causes that they must be artificially stored as (one element) collections. This violates conceptual modeling and is error-prone. Lack of collections in programming languages causes that the programmer is forced to use lower-level mechanisms such as heaps, with obvious disadvantages (no stack discipline, dangling pointers, memory leak, etc.). Summing up, nowadays there is little justification for the assumption that persistent data are to be collections, volatile data are to be individual and typing systems of persistent and volatile data should be different.

The orthogonal persistence principle requires that the persistence property is orthogonal to kinds of data (or objects). In particular, a database can store individual objects (not only collections) and the volatile main memory of an application can contain collections of objects. Moreover, orthogonal persistence requires that the type system is the same for persistent and volatile data/objects. Looking at query languages that we would like to develop, there should also no syntactic and semantic differences in access to persistent and volatile data.

Unfortunately, orthogonal persistency is against the tradition in software industry, thus some professionals attack it as impractical. In our opinion such attacks have ideological rather technical foundations (mostly caused by attempts to defend current commercial software products). From the technical point of view it is difficult to imagine why orthogonal persistence would result in some critical disadvantages.

10.9 Data Independence

Data independence means that a database is designed and maintained independently of applications that retrieve and manipulate the data. There are several levels of data independence. Physical data independence means that physical details of data organization and access are transparent for the application programmer. The physical details are
determined by the database designers or even earlier, by the designers of a database management system. Some physical details (e.g. indices supporting the access to data, special file organization, special methods of performing operations, etc.) are under control of a database administrator (DBA). DBA uses special administration module to tune the database operation according to the actual demands of applications, but still, nothing in applications has to be changed due to the tuning. Logical data independence means that DBA is able to perform some operations on the database structure, for instance, add new data kinds, add or remove some object or table attributes, change user privileges, add and remove views, database procedures, triggers etc. without unconscious influencing the applications. Conceptual data independence means that DBA is able to change the structure of the database conceptually without changing existing (legacy) applications, for instance, through special wrappers, mediators, views, updatable views, do instead of triggers, and other means that allow to change significantly the database schema and its organization, perhaps with minor changes of applications. The last level of data independence is referred to as schema evolution and conceptually is close to software change management methods and the aspect-orientation in databases.

Data independence means that a query language has no notions referring to physical details of data organization such indices. Logical and conceptual data independence requires that a query after compilation is still correct even in case of significant logical or conceptual changes in the database. This implies that the compiled query has no elements dependent on physical details, such as offsets of attributes within objects or tuples. This usually leads to the necessity of run-time (late, dynamic) binding of all names occurring in a query.
11 Appendix 2. Impedance Mismatch

Impedance mismatch\(^{17}\) is perceived by many professionals as a negative phenomenon in the software construction that arises from an eclectic mix of two incompatible languages: a query language that is used to access and update of a database and a programming language that is used for making client applications acting on the database. A bit more careful look shows that this negative connotation is perhaps inadequate. Impedance mismatch is an inevitable consequence of a (quite reasonable) principle known as data independence. Misunderstanding of the relationships between impedance mismatch and data independence is the reason of some creativity within modern object-oriented programming languages (mostly Java), which some professionals could perceive as a medicine that is worse than the illness. Impedance mismatch has also less discussed aspect concerning database models and transformations between database schemas. In the following we will try to discuss all the above mentioned aspects.

11.1 Impedance Mismatch between Query and Programming Languages

The concept of query languages developed in 1970-ties assumed no pragmatic universality. However, because eventually such universality is inevitable in real applications, there was an assumption that a query language is a “sublanguage” that is to be embedded in a universal programming language. A “sub-language” determines the access to a database only. The rest of the entire application has to be programmed in a typical programming language. This assumption requires joining a query language with a programming language in such a way that:

- Queries can be used inside programs;
- Queries can be parameterized through values calculated within programs;
- Results of queries are to be passed to programs.

Difference between language concepts cause significant technical difficulties in accomplishing this kind of connection. A lot of programmers and computer professionals were also disappointed by the technical, aesthetic and conceptual degradation of the programming environment. This degradation is commonly referred to as impedance mismatch. This term denotes a bunch of disadvantageous features that are implied by the eclectic mix of a query language (in particular SQL) with a programming language (such as C, C++ or Java). Below we list and comment these features.

- **Syntax:** In the same code the programmer must use two programming styles and must follow two different grammars. Similar concepts are denoted differently (for instance, strings in C are written within “…”, in SQL – ‘…”’) and different concepts are denoted similarly (for instance, in C = denotes an assignment, in SQL – a comparison).

- **Typing:** Types and denotations of types assumed by query and programming languages differ, as a rule. This concerns atomic types such as integer, real, boolean, etc. Representation of atomic types in programming languages and in databases can be significantly different, even if the types are denoted by the same keyword, e.g. integer. A

\(^{17}\) The term impedance mismatch has roots in electronics, where it denotes the difference between the impedance of a source and the impedance of a receiver, causing that the effective power is wasted.
lossless conversion between such types could be impossible and might imply some performance overhead. This also concerns complex types, such as tables (a basic data type constructor in SQL, absent in programming languages). Popular programming languages introduce static (compile time) type checking, which is impossible e.g. in SQL (because query languages are based on dynamic rather than static binding).

- **Semantics and language paradigms**: The concept of semantics of both languages is totally different. Query languages are based on the declarative style \( \textit{what} \) is to be retrieved rather than \( \textit{how} \), while programming languages are based on the imperative style (a sequence of commands to a machine, which accomplishes \( \textit{what} \)).

- **Abstraction level**: Query languages free the programmers from majority of the details concerning data organization and implementation, for instance, organization of collections, presence or absence indices, etc. In programming languages these details usually are explicit (although may be covered by some libraries).

- **Binding phases and mechanisms**: Query languages are based on late (run-time) binding of all the names that occur in queries, while programming languages are based on early (compile and linking time) binding. Thus, from the point of view of a program, queries are simply strings of characters.

- **Name spaces and scope rules**: Queries do not see names occurring in programs and v/v. Because eventually there must be some intersection of these name spaces (e.g. program variables must parameterize queries) additional facilities, with own syntax, semantics and pragmatics, are required. These facilities are the burden for the size and legibility of the program code. Moreover, in programming languages name scopes are organized hierarchically and processed by special rules based on stacks. These rules are ignored by a query language. This leads e.g. to problems with recursive procedure calls (a well-known example concerns SQL cursors that severely reduce the possibility of recursion). Another disadvantage of separated name spaces concerns automatic refactoring of programs, which cannot be performed on queries.

- **Collections**: Databases store collections (e.g. tables) which are processed by queries. In programming languages collections are absent or severely limited. Hence collections returned by queries have no direct counterparts in a programming language and must be processed by special constructs with own syntax and semantics.

- **Null values**: Databases and their query languages have specialized features for storing and processing null values. Such features are absent in programming languages, thus some substitutes must be introduced. For instance, if some value in a relational database can be null, mapping it to a programming language requires two variables: one for storing information about null and another one for storing the value.

- **Iteration facilities**: In query languages iterations are accomplished as macroscopic query operators, such as selection, projection, join, product, sum, intersect, etc. In programming languages iterations are coded explicitly as loops (\textit{for, while, repeat}, etc.). Processing results of queries in a programming language requires special mechanisms, such as cursors and iterators.

- **Persistence**: Query languages address persistent data (stored on a disc or another long-term memory), while programming languages process only data stored in a volatile operating memory. Joining query and programming languages requires special facilities for parameterizing queries by volatile variables and transmission of persistent data to volatile memory and v/v.
• **Queries and expressions:** There is some competence mismatch between queries and programming language expressions. Some queries look as expressions and v/v, but there is strong syntactic subdivision of them, which can be poorly understood by the programmer. For instance, in some query languages 2+2 is a query, but it is also an expression of a programming language. A query cannot be a parameter to a procedure, but an expression can. There could be other syntactic constraints, which cause a lot of chaos in the entire programming environment.

• **References:** If a query is to be used for updating, inserting or deleting, it must return references to stored data (i.e. data identifiers rather than values). According to the official semantics of the relational model, queries return tables of values, with no references. For updating constructs defined in a programming language such semantics is inconsistent; actually, it means that queries cannot be used for updating or require a special mode of execution and/or a special constructs in a programming language.

• **Refactoring:** decisions concerning new names used for data structures cannot be automatically propagated to queries, because – from the point of view of a programming language – queries are strings, sometimes not explicitly seen from the source program. Hence refactoring of queries should be done manually, with a lot of effort and possibilities of inconsistencies.

The consequences of impedance mismatch concerns not only aesthetics and ergonomics of the programming environment. Impedance mismatch implies an additional programming layer, with own syntax, semantics and pragmatics. This layer causes that learning of the language takes more time, programming is more error prone, and programs are unnecessarily longer and less legible. This layer may also cause worse performance and maintainability of applications. If queries are strings then there is no explicit support for creating reusable query components. None of the reusability features of the programming language (functions, methods, and polymorphism) are available to support reuse. Passing parameters to queries written as strings (c.f. the ODMG standard) is awkward and error-prone. Queries, as strings embedded in a program, are also more prone to injection attack.

Some authors suggest that the source of impedance mismatch is in incompatibility of data models, in particular, access to relational databases is accomplished from an object-oriented programming environment (such as object-oriented DBMS). Such a suggestion presents the ODMG standard, with the conclusion that in this standard impedance mismatch no more holds. Unfortunately it is only partly true. Indeed, the mismatch is inevitable in situation of big differences between data models, in particular, between a relational system and an object-oriented programming language. However, even if both models are claimed to be “object-oriented”, the impedance mismatch still persists. There are significant differences between various object-oriented models. Actually, there are as many object models as different object-oriented artifacts and proposals; no standard object model exists. Differences between the object models of Smalltalk, CORBA, UML, C++, SQL-99, ODMG, Java, C# etc. are fundamental. Moreover, even if the model is exactly the same and would the subject of some precise standard, some impedance mismatch can persist due to e.g. differences in binding phases. These issues we discuss in the next subsection devoted to the relationship between impedance mismatch and data independence.

To avoid the impedance mismatch, a language should be integrated, where queries are smoothly connected with programming constructs and abstractions. This tendency is seen in such products as PL/SQL, T-SQL and standards SQL-99 and SQL-2003. The SBQL language is designed according to this tendency, where SBQL queries are integrated with updating constructs and programming abstractions. The typing system and a strong type checker is the
same for the queries, updating constructs, procedures, functions, methods, parameters, views, transactions, etc. This significantly distinguishes SBQL from other query languages. In the following we discuss some pros and cons of the idea of integrated database programming languages.

11.2 Impedance Mismatch and Native Queries

Recently there are some proposals known as native queries [Cook06] that support some kind of queries within typical programming languages such as Java (db4o) or C# (LINQ [Linq07, Linq10] and Comega [Come07]). This mechanism is advocated as 100% typesafe, 100% compile-time checked and 100% refactorable. Unfortunately, such an approach introduces limitations to queries, in particular:

- Queries dressed in the syntax of a programming language are much less legible even in comparison to SQL (which in our opinion is not much legible too). There is a lot of information noise from the point of view of direct expressing of a query goal.
- The expressive power of queries is far below the acceptable power, for instance, in native queries there are no joins, grouping, aggregate functions, sorting, quantifiers, set operators, etc. Actually, such queries cover only simple selections.
- It is impossible to formulate ad hoc queries, because of early binding.
- It is impossible to have server-side persistent abstractions based on queries, such as database views, stored procedures, triggers, etc.
- The problem with native queries concerns performance for very large collections of objects. In [Cook06] the problem is recognized (and claimed to be solved through changes to a compiler), but the solution seems to be not enough flexible. Native queries are very simple; more complex tasks must be accomplished by a sequence of several queries, instead of just one. Because such a sequence is mixed up with the regular code, generating finally a single composed query is practically impossible. This much reduces opportunities for optimization. In particular, essentially no optimization by rewriting rules is possible.
- Query optimization is not directly supported by native queries – they rely on SQL optimizers. To utilize an SQL optimizer the mapping between objects and relations cannot be too complex; otherwise the developers have to solve non-trivial research problems, similar to the view updating problem that is commonly recognized in the database domain as very difficult. This limitation can much warp the initial conceptual object data model. In majority of cases this leads to almost trivial mappings between Java or C# objects and relational tables. Then, mapping between native queries and SQL is equally trivial. In effect, native queries still work on the relational data model. In this way the object model becomes a “slave” of the relational model. Although relational tuples are wrapped as “objects”, this is far from true object-orientedness.

The idea of native queries can work for some kinds of applications, but not for real very large databases. The LINQ project has a bit better position, because it assumes extensions of the C# and VB syntax. This approach can be perceived as a reverse approach in comparison to SBQL: LINQ extends the syntax and semantics of a programming language in direction of query languages, while SBQL attempts to extend a query language in the direction of a programming language.
Java and C# were developed without database programming in mind. Extensions to these languages in this direction are of course possible (as proponents of JDO, Hibernate, LINQ, native queries, etc. have shown), but with a lot of limitations and awkward solutions. Only integrated database query and programming language, developed with taking into account all database peculiarities, is able to fulfill all critical requirements to such software manufacturing environments. Native queries present too little (if any) progress in this direction.

11.3 Impedance Mismatch and Data Independence

Impedance mismatch is an inherent consequence of the data independence principle that assumes that a database is designed, administered, maintained, secured, catalogued, published and accessed independently from any application program that acts on the database. Moreover, there is usually no assumption that database applications are to be written in a single programming language. Just otherwise, data independence implicitly assumes that a database will be available for many programming languages, providing each of them implements a corresponding library (or a “driver”). Integrated languages that make no distinction between querying and programming violate data independence. In most common situation the idea of an integrated language implicitly assumes that database application programs will be written in a single programming language (or in a family of languages having the same typing and/or data representation system). It is not sure if the software community is currently prepared for such a “monopoly” (which apparently violates our sense of democracy, free commercial competition, free possibilities of inventions and the need of diversity as a progress factor). However, as a matter of fact, many database applications are actually based on such monopoly. It is possible of course that each database management system would work with its own integrated programming language; PL/SQL of Oracle and T-SQL of SQL Server are examples. However, when one tries to connect database servers of different types (or even different cultures) is a single application then impedance mismatch seems to be inevitable. Actually, impedance mismatch can be avoided by well-developed wrappers to external resources. However, wrappers introduce a lot of limitations (data models, performance, security, transaction processing, etc.) hence might not be the best for a lot of cases. We discuss this problem a bit later.

In the relationships between impedance mismatch and data independence there is no ideal solution. In particular, a tradeoff is necessary for the data independence principle. The principle was formulated at the time when databases (especially relational databases) contained pure data only. Current database servers, including relational ones, store many entities that must be prepared in a query and programming language. These entities include:

- Stored procedures and functions.
- Triggers, constraints and (business) rules.
- Stored classes, including methods that are defined within these classes, inheritance, and other features of object-orientedness.
- Database views, in particular, updatable database views.
- Definitions of workflow processes.
- Definitions of wrappers, mediators, adapters, integrators, exporters, importers and other interoperability or data distribution facilities.
These (persistent) entities are prepared during the database design phase or during database maintenance by a database server administrator. Some other entities are possible and are currently considered such as persistent threads, pre- and post-conditions, assertions, overloading views and so on. One can imagine that these entities can be written in many languages, but for several reasons such a freedom would be disadvantageous or unrealistic. All such languages should be based on the same data structures (determined by the database model and types) and this limitation much reduces the freedom. The assumption that any programming language can be used for this purpose is unrealistic at least for two reasons: (1) early binding assumed in popular languages (which would exclude many database features such as views, changes in the database schema, etc.); (2) severe problems with impedance mismatch. Hence, as a final conclusion, for a given DBMS all such active entities should be written in a single, integrated query and programming language that deals with persistence as a regular option. For these reasons the development of integrated database programming languages and their standards makes a great sense. The ODRA project has shown that within this solution some level of interoperability with external incompatible databases (with no impedance mismatch) can be achieved by properly constructed and implemented wrappers.

11.4 Impedance Mismatch between Models and Schemas

The impedance mismatch phenomenon is not limited only to incompatibilities between a query language and a programming language. In the literature another form of impedance mismatch is discussed: the mismatch between an object-oriented conceptual schema and an equivalent relational schema. This mismatch is even more disadvantageous for the entire software life cycle than the previous one. To a big extent, just this kind of mismatch is the greatest motivation for object-oriented databases.

We illustrate the mismatch on an example. In Fig.A2.1 we present a UML-like object oriented-database schema that is the result of the analysis and design phases of some application. Such a schema can be directly written in the ODRA system.

![Object-oriented UML-like database schema (ODRA)](image)

For simplification in the schema we omit atomic types considering them as obvious. The schema specifies obvious dependencies between persons, experts and companies, including cardinalities of particular data. An expert has some set of competences. He or she can do many works for a company; each work may require many payments and many expertises. The expert class inherits from the person class. A person can have many names and addresses. A company can have many locations. Experts, works and companies are connected by two obvious associations. Not that this schema is very close to UML, but with essential differences that (we hope) do not disturb smooth, seamless mapping between an UML class diagram to the database schema. In the schema cardinalities are assigned to all data types.
(cardinality [1..1] is omitted). Associations can be only binary and association role names are substituted by pointer types (EW, WE, WC and CW).

According to our experience, an average programmer can understand this schema in few minutes, and then he or she can write SBQL queries and programs. If an object database management system is unavailable, then this schema must be transformed to the relational schema. One of possible solutions is shown in Fig.A2.2.

![Diagram](image)

**Fig.A2.2.** Relational schema “equivalent” to the object schema from Fig.A2.1

Now we clearly see that the “simplicity” of the relational schema is apparent. The programmer must spend a lot of time to understand it. Perhaps, he or she would require additional informal information. One of possible ways to give this information is to refer to the original UML design diagram a similar ER diagram. (Just a version of such diagrams is presented in Fig.A2.1.) In the diagram presented in Fig.A2.2 some conceptual information is lost. This concerns, in particular, boundaries of objects and cardinalities. For instance, the information that each company must possess at least one location cannot be inferred from Fig.A2.2. The diagram is also littered by some inessential information that in the relational representation must be explicitly filled in: for instance, originally we have no C#, E#, P# and W# attributes – they have appeared as the result of the relational normalization. Arrows that lead from foreign to primary keys are actually not a part of the relational schema – they are depicted as obvious but may become much less obvious in case when (for various reasons) the names of foreign keys are different from the names of primary keys. This means that the relational schema is much more complex in comparison to the equivalent object-oriented schema.

SBQL is one of few query languages that is prepared to address queries against to both object-oriented and relational schemas. Now we can observe how this case of impedance mismatch between schemas will influence the complexity of queries:
**E.A2.1**  Get last names of experts that prepared at least 3 expertises for IBM:

<table>
<thead>
<tr>
<th>SBQL:</th>
<th>The object schema (26 lexical units):</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Company where cName = &quot;IBM&quot;) . CW . (Work where count(Expertise) &gt;= 3) . WE . Expert . lName</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SBQL:</th>
<th>The relational schema (70 lexical units):</th>
</tr>
</thead>
<tbody>
<tr>
<td>((Person as p, Expert as e, Work as w, Company as c) where p.P# = e.P# and e.E# = w.E# and c.C# = w.C# and c.cName = 'IBM' and count(Expertise as e where w.W# = e.W#) &gt;= 3). P.lname</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQL:</th>
<th>The relational schema (72 lexical units):</th>
</tr>
</thead>
<tbody>
<tr>
<td>select p.lName from Person as p, Expert as e, Work as w, Company as c where p.P# = e.P# and e.E# = w.E# and c.C# = w.C# and c.cName = 'IBM' and (select count(*) from Expertise as e where w.W# = e.W#) &gt;= 3</td>
<td></td>
</tr>
</tbody>
</table>

This example presents a schema impedance mismatch rule that can be formulated as follows:

| **More semantics in a schema implies shorter and more comprehensive queries.** |
| **Less semantics in a schema causes longer and illegible queries.** |

The rule presents a great argument in favor of object-oriented database models. Some estimations have shown that all the cases of impedance mismatch cause that the size of application code addressing a relational database is about three times longer than the code addressing an original object-oriented database schema. This is obviously disadvantageous concerning time, money and quality of applications. The most disadvantageous is the cost of maintenance, which frequently requires reverse engineering in which the programmer must return to the original object-oriented analysis and design schema.

Another aspect of the above impedance mismatch case concerns relationships to object-relational mappers (ORM) such as Hibernate. The problem that we see is that the basis for an object-oriented schema in ORM is the relational schema rather than the original UML-like schema. Hence the question is what is the relationship between the final ORM database schema that is seen e.g. to Hibernate programmers and the initial UML schema that was the result of analysis and design phases. Because of several problems with ORM (in particular, SQL-based query optimization) we can suspect that ORM schema is rather trivial mapping of the relational schema and has little in common with the original UML schema. This means that the impedance mismatch persists when the programmer (for any reason) has to reconstruct the original UML schema. This could be especially painful and frustrating for the maintenance activities which frequently require referring to the original analysis and design documentation. Unfortunately, the Author of this report does not know any serious research that would investigate how many effort (or time, money, quality decrease, frustrations of programmers, etc.) is caused by this case of impedance mismatch.

### 11.5 Mediators, Adapters, Wrappers and Virtual Repositories

[Wied92] proposes the technology of *mediators* that seem to be a method to avoid impedance mismatch. A mediator adapts an external resource (incompatible in the sense of data model, semantics and access) to the requirements of the given applications. A similar idea (accomplished on a lower abstraction level) is accomplished in CORBA and object request
brokers based on the standard. The corresponding terms are *adapter* and *skeleton* that change the application programming interface of some external resources to the interface determined an expression of IDL (Interface Definition Language). In this way CORBA accomplishes transparent access to distributed heterogeneous resources which can be implemented on different sites, media, with different representation of data, on different systems, etc. The idea is also known under the name *wrapper*, i.e. some software that converts an API (Application Programming Interface) into another API that is required by some canonical data model. There are other ideas that correspond to the previous ones, such as Web Services and Service Oriented Architecture (SOA).

More recently, the idea of mediators, adapters, wrappers, etc. is generalized under the name *virtual repository*. In a virtual repository the source information is not replicated or stored. The information is accessed from multiple applications by using some kinds of links or mappers that redirect or map an original request to the request addressing particular data and service sources. After the request is accomplished, its result is mapped back from an original source format to the format expected by the given application. Hence a virtual repository presents a transparent middleware layer (similar to a CORBA bus) that makes it possible to write homogeneous programs (e.g. in Java) addressing distributed and heterogeneous resources without involving in the programs any peculiarities or details related to organization or access to the resources. Idealistically, in this case the impedance mismatch no more holds, as there is no necessity to mix two incompatible languages in one source code. The entire job with the access to external resources is done during the middleware building phase, earlier than any application code is to be written. The application programmer uses external resources in such a way that they are non-distinguishable from regular data structures (objects) in a regular working space of the application. In another parlance, peculiarities of external resources organization and access are transparent for the user of a virtual repository.

In the ODRA project and in the European project eGov Bus we have collected a lot of research and experience concerning the idea of virtual repositories.
Fig. A2.3. Reference architecture of a virtual repository (ODRA)

Fig. A2.3 shows some architectural variant for a Virtual Repository that we can be developed in the ODRA system. It presents some configuration of developed software units. Many other architectural combinations are possible, depending on a particular application in question.

A central part of the architecture consists of ODRA, an object-oriented DBMS. Existing resources (bottom of the figure) are extended by wrappers and contributory views (or importers/exporters) that convert data/services proprietary to particular existing applications into the format acceptable for ODRA. The application developers can install as many ODRA servers as necessary, addressing the same distributed sources. The integration view on the ODRA server allows for virtual integration of data and services supplied by distributed sources, supporting data and function abstractions. The virtual repository front-end will provide various APIs to access virtually integrated data, including workflow applications, Java applications, Web services applications, and others. These APIs are available from SBQL, a query and programming language of the ODRA system. A particular user works with own client view that is a tailored part of the entire virtual repository schema. Among many other functions, the virtual repository allows for transparent access to external information resources and for unlimited transformations of complex document structures.

To accomplish such architecture without the impedance mismatch effect several conditions must be satisfied, in particular:

- Front-end data model (seen by application programmers) must be rich enough to cover conceptually all the data models of external sources. If the front-end data model is limited, some structures within existing sources have no direct mapping, hence must be supported by many commands in a front-end programming language. This is a kind of impedance mismatch. For instance, if any source supports collections, then the front-end data model must support collections too.
• Front-end query and programming language must be at least on the same abstract level as any of application programming interfaces of sources. For instance, if a source is accessible via a query language, then the front-end language must support a query language at least as powerful as the source query language. Otherwise the power of the source query language cannot be utilized in the front-end language, which implies (at least) severe performance problems and many statements of the front-end language that are needed for expressing one query in the source. This is another sort of impedance mismatch. Such impedance mismatch is inevitable if one assumes Java as a front-end language and SQL–based sources (independently whether native Java queries or embedded query strings are used).

• Mapping between a source data and a front-end virtual data must be done by a tool that is sufficiently powerful. If the mapping capabilities are restricted, then the front-end schema is much burdened by source schemas, hence the degree of abstraction and transparency is much decreased. In general, all the mapping freedom is available only in the case when the mapping tool has the algorithmic power of universal programming language. Other mapping tools (e.g. some special XML files) in any case introduce a lot of limits.

• The mapping should utilize native query optimizers that are implemented in source databases (e.g. rewriting rules or indices). Otherwise the performance is usually unacceptable. In particular, a front-end query and programming language should give all the chances for SQL optimizers to work. Unfortunately, tests on some ORM solutions show significant performance problems. Bad performance undermines scalability, thus is unacceptable for majority of business applications.

• Mapping is relatively easy when one considers data retrieval only. For mapping of updates that act on front-end virtual objects and result in updating of sources the mapping capabilities can be very complex, up to programming nightmare. The problem is recognized in the database literature as the view updating problem. For this reason mapping of updates severely restricts the conceptual distance between source schemata and the target (front end) schema. In majority of solutions (especially concerning ORM) the mapping is practically isomorphic (1:1), i.e. trivial and hardly acceptable from the point of view of object-oriented conceptual modeling.

• There are many challenging issues during implementation of such a virtual repository, such as mapping between incompatible atomic types, mapping of procedural capabilities implemented on the side of sources, global indexing of data existing in heterogeneous distributed sources, distributed query optimization, replicated and redundant data existing in heterogeneous distributed resources, distributed transaction processing, mapping of administrative facilities (such as security and user privileges), etc. No general approach exists to solve them; each peculiarity of the mapping must be solved by a proprietary method.

In ODRA the mapping between source and target schemata is done by object-oriented virtual updatable views. In contrast to many other proposals of views, ODRA views are defined in SBQL, which has full algorithmic power. There is no conceptual limit concerning the mapping. This especially concerns ORM. SBQL queries addressing an object-oriented front-end schema are modified by view definition (by the query optimization technique) thus are converted into SBQL queries addressing the corresponding relational database. Then, these queries are optimized and converted into SQL. Due to this technique no form of impedance mismatch (from discussed above) can occur on the front-end level. Mapping of updates of virtual objects into updates of source structures can also be implemented within ODRA updatable views, without any conceptual anomalies and warping the updating intention.
(which plagues a lot of solutions concerning views). However, this issue is more critical than pure retrieval, especially concerning performance. Because for majority of business applications performance dominates over clean conceptual model, elegance of solution and minimal programming effort, solving performance problem always means introducing some impedance mismatch from the kinds that are discussed above.
References


[Drabik09] M. Dąbrowski, M. Drabik, P. Habela, K. Subieta: *Object-Oriented Declarative Workflow Management System*. Editors of the Institute of Computer Science, Polish Academy of


[Date86b] (In [Date86], Chapter 14) A Critique of the SQL Database Language.

[Date86c] (In [Date86], Chapter 15) Null Values in Database Management.


[Date92b] (In [Date92], Chapter 18) Oh No Not Nulls Again.

[Date92c] (In [Date92], Chapter 21) The Default Values Approach to Missing Information.


Liu, A Recursive Object Algebra Based on Aggregation Abstraction for Manipulating Complex Objects. Data & Knowledge Engineering 11(1), 21-60, 1993


Object Management Group: UML 2.0 superstructure specification (pct/03-08-02), 2003


D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, J. Widom. Querying Semistructured Heterogeneous Information. Proc. of the International Conference on Deductive and
Object-Oriented Databases (DOOD95) Springer Lecture Notes in Computer Science 1013, 1995, pp.319-344 (http://www-db.stanford.edu/pub)


[Subi10] K. Subieta. Stack-Based Architecture (SBA) and Stack-Based Query Language (SBQL). http://www.sbql.pl/, 2010


13 List of Figures

Fig.1.1. Database schema and database structures complying to the schema .......................................................... 17
Fig.1.2. AS0 model - a small database ................................................................................................................ 30
Fig.1.3. AS0 model - a graphical representation of a small database ..................................................................... 31
Fig.1.4. Relational database represented in the AS0 store model ......................................................................... 36
Fig.1.5. XML file and its AS0 representation ....................................................................................................... 38
Fig.1.6. XML file with attributes and its equivalent without attributes ................................................................. 39
Fig.1.7. XML file mixing strings and objects and its equivalent with no mix .......................................................... 39
Fig.1.8. Example of an AS1 object store ............................................................................................................... 44
Fig.1.9. Graphical representation of the example AS1 object store ...................................................................... 45
Fig.1.10. Roles of a person object ......................................................................................................................... 46
Fig.1.11. Example of an AS2 object store .............................................................................................................. 48
Fig.1.12. Graphical representation of the example AS2 object store ....................................................................... 49
Fig.1.13. Example of an AS3 object store extending an AS1 object store by export lists .................................. 55
Fig.2.1. Example situation on the environment stack ......................................................................................... 58
Fig.2.2. Sample state of an object store .............................................................................................................. 63
Fig.2.3. A sample state of an environment stack ENVS ..................................................................................... 63
Fig.2.4. Order of searching ENVS during the binding a name ........................................................................... 65
Fig.2.5. Query results depicted as tables ............................................................................................................. 68
Fig.2.6. States of the arithmetic stack during calculation ................................................................................... 69
Fig.2.7. Example of a QRES state ....................................................................................................................... 69
Fig.2.8. Pushing a new section on ENVS to evaluate a condition ...................................................................... 70
Fig.2.9. Illustration of the function nested ....................................................................................................... 72
Fig.2.10. Dependencies between object store, environment stack and query result stack ............................... 73
Fig.2.11. Architecture of query processing with strong type checking and query optimization .................. 74
Fig.3.1. SBQL basic syntax ................................................................................................................................. 80
Fig.3.2. QRES changed by the procedure eval and by a query result consumer .............................................. 82
Fig.3.3. Syntactic tree of the query: Emp name = “Poe” .................................................................................. 82
Fig.3.4. Schema of evaluation of a query involving a non-algebraic operator ................................................... 98
Fig.3.5. Steps of evaluation of the query Emp sal > 1000 ................................................................................ 100
Fig.3.6. Relational schema and an equivalent object schema used in examples ............................................... 104
Fig.3.7. ENVS during evaluation of the query Dept join avg{employs.Emp.sal} ................................................ 109
Fig.3.8. Schema used in examples .................................................................................................................... 112
Fig.3.9. Class diagram and ENVS during processing object O by a non-algebraic operator ......................... 124
Fig.3.10. State of ENVS during processing the Poe's object by the dot operator .......................................... 124
Fig.3.11. Evaluation of a method ....................................................................................................................... 127
Fig.3.12. Database schema according to the AS1 store model ........................................................................ 130
Fig.3.13. Beginning state of ENVS in the model with dynamic roles ............................................................. 132
Fig.3.14. State of ENVS during processing a role Emp by a non-algebraic operator ....................................... 133
Fig.3.15. Object with five roles connected to their classes ................................................................................ 134
Fig.3.16. Situation on ENVS when the role R5 is processed by a non-algebraic operator .......................... 134
Fig.3.17. A database schema for the model with dynamic object roles .......................................................... 136
Fig.3.18. Conceptual and optimized environment stack ................................................................................... 150
Fig.3.19. A database schema ............................................................................................................................ 154
Fig.3.20. ENVS stack: procedure P1 calls procedure P2, which calls procedure P3 ...................................... 167
Fig.3.21. A sample recursive database schema ................................................................................................. 175
Fig.3.22. A sample structure for the examples of an application of close unique by .................................. 178
Fig.3.23. Example of a graph of parts .............................................................................................................. 181