OBJECT-ORIENTED QUERY LANGUAGE SPECIFICATION AND QUERY PROCESSOR GENERATION

by

Jibin Zhan
B. Sc., Peking University, China, 1988

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
June 1994

ISBN 0-612-06882-X

APPROVAL

Name:	Jibin Zhan
Degree:	Master of Science
Title of thesis:	Object-Oriented Query Language Specification and Query Processor Generation

Examining Committee: Dr. Hassan Ait-Kaci
Professor, Computing Science
Chair

Dr. Wo-Shun Luk
Professor, Computing Science
Senior Supervisor

Dr. Jiawei Han
Associate Professor, Computing Science
Supervisor

Dr. Warren Burton
Professor, Computing Science
Examiner

Date Approved: June 17, 1994

Abstract

In this thesis, we proposed a systematic approach for query language customization which can lead to Customized Query Languages (CQL) and different query front ends on new generation database systems in the heterogeneous database environment. Our approach is based on a core Object-Oriented data model and query model that is derived from the data models of Object Management Group(OMG, 1990), Object Database Management Group(ODMG-93) and some other Object-Oriented and Extended Relational Database Systems, like O2, Orion, Postgres, and SQL3. Object-Oriented methodology is used in designing and presenting the data model and query model, i.e. all the basic components in the data model and query model are abstracted into meta-objects and meta-types, for which some basic characteristics and operations are defined. Special default "constructors" for all the meta-types are provided to represent both the default semantics and syntactic appearance of the corresponding components in our default Object Definition Language(ODL) and Object Query Language(OQL). Query language specifiers can provide their own "constructors" for these meta-types to override the default ones in order to tailor the syntactic appearance and/or semantics. New components and operations corresponding to some new components in the specified query languages can also be defined in a similar way as the new types and functions defined in C++. Following this approach, a non-procedural specification language is proposed which leads to automatic generation of query language specific processor in LEX and YACC.

Acknowledgements

I would like thank to my senior supervisor, Dr. Wo-Shun Luk, for his encouragement, guidance, fruitful discussions and many other helps. Without his full support, this thesis will be impossible. I would also like to thank Carlos Wong for his productive discussions on my work.

Special thanks to Andrew Fall and Martin Vorbeck for their patient proof-reading of my thesis.

Contents

List of Figures

Chapter 1

Introduction

1.1 New Generation Database Systems

Relational Database Systems, which mainly focused on business applications and only provided simple data management have been proved inadequate for many advanced applications, such as CAD(Computer Aided Design), CASE(Computer Aided Software Engineering), GIS(Geographic Information System), etc. In the past decade, a lot of research and development have been conducted and tried to bring the so-called new generation database systems to these application areas.

There are several different approaches based on different data models to address the problems of new generation database systems. The most obvious approach is based on the extended relational data model which incorporates complex data types, procedures and other new features into the basic relational data model. This leads to the extended relational database systems, for example, Postgres. Another approach is actually inspired by the Object-Oriented Programming Languages. The data models, which have complex data types and operations defined on the data types, are called object-oriented data models, and the systems based on these models, such as O2, Orion, Objectstore, etc are called Object-Oriented Database Systems. Most of the current new generation database prototypes or commercial systems are based on these two kinds of data models, although there are some others based on functional data models or semantic data models.

Despite these different approaches, "There is a surprising degree of consensus on the desired capabilities of these next-generation systems" [17]. As summarized in [17], [4] (Chapter 6), etc, one of the primary goals of the new generation DBMS is to support complex data management. This means that, first the DBMS should provide facilities for directly describing complex data structures in a natural way for the applications; and secondly the DBMS should also provide an efficient and natural way to store and manipulate the complex data. From users' perspective, this means that the DBMS should provide an interface, usually in the form of query languages, either declarative or procedural, stand-alone or embedded, that can deal with complex data.

1.2 Query Language Issues for New Generation Database Systems

1.2.1 Declarative or Procedural

Standard non-procedural query languages are one of the main reasons for the success of relational database systems. For the new generation database systems, people begin to agree that declarative data definition languages and data manipulation languages are still one of the most desirable features to the users. This is not only because declarative languages are easy to learn and use, but also they can provide more physical independence. In a lot of typical situations, performance of high-level queries can be better than ordinary hand coded low level associative queries. This is because the system itself knows more details of its implementation and there are a lot of well-defined query optimization algorithms for these cases. This is quite similar to the fact that people usually do not have to write programs in assembly languages if there is a good compiler for a high-level language.

1.2.2 Standardization

For relational database systems, the standard query language is based on the simple relational data model, relational algebra or relational calculus, in which there are only one structured type(relation) and three operations(selection, projection, and join), which satisfy the *database computational completeness* [4] (i.e. in some sense, all the computational tasks that on the information in the database can be done with these operations). We can say that the origin of the relational query language is the relational data model theory.

Unlike RDBMS's(Relational Database Management Systems), the origin of declarative query languages is the application needs and the compatibility with SQL. But in reality, there are a lot of difficulties in providing a general declarative query language for all the new generation database systems. One of the main reasons is that there is no single data model that is widely accepted for the new generation DBMS. Because of the complexity of the underlying data model, there are also a lot of variants even for the same data model. For all these data models, none of them has a very solid theoretical foundation that is similar to that of the relational data model. As a result, the query languages for the new generation database systems are quite different from one to another. Right now, no standard query language for the new generation database systems are widely accepted like SQL for RDBMS. Instead, different systems provide their different query facilities, and some of the object-oriented database systems even have not provided any declarative query language at all.

1.2.3 Customization of Query Language

Another problem related to the declarative query languages is that for many advanced applications, there are a lot of application specific semantics in the data. When users do the data entry, maintenance and manipulation, they want to use a query language that embeds a lot of these semantics in it, so that it is much easier and convenient for them to use. They may even want to use the vocabularies specific to the applications in the query languages, not only just the ordinary select-from-where format. These kinds of query languages are called Customized Query Languages(CQL). CQLs usually are defined and implemented by some application programmers on a certain underlying platform. Usually, to provide a clearly defined and efficiently implemented CQL is not an easy work for ordinary application programmers.

Although many end users may prefer CQL for many advanced applications and they may not want to use any general query facilities, the need for CQLs should not be used as an argument against the need for general declarative query languages. This is because many end users and application programmers still need query languages to phrase their ad hoc queries. CQL and the general query language are actually in different layers of the database systems. As we can see later, usually a CQL should be built based on the basic components of the general query languages instead of starting from scratch.

1.3 Current Research

There is a lot of research on query languages for the new generation database systems. For the extended relational database systems, they mainly focus on the extension of relational query languages by incorporating a lot of object-oriented concepts into them. For example, Postquel(which is based on relational query language Quel of Ingres) in Postgres. For the object-oriented database systems, they usually want to adopt SQL-like format for some typical data manipulation patterns because of the popularity of SQL. Several systems like O_2, ORION, etc, provide stand alone query facilities in the format similar to the standard SQL; and some others like Objectstore, provided embedded declarative and/or associative query languages. Although, the semantics and functionalities of the query languages are quite similar, there are still a lot of differences between query languages of different systems.

The variants of the new generation systems and their query languages create a lot of problems for the users. At the current stage, with a lot of network connections and different database systems co-existing, users may have to learn different query languages for different database systems. When they phrase the queries, they have to know what kind of database systems they are using. If they want to use customized query languages, then the application programmers may have to implement the language on different platforms, which may need a lot of expertise of different database systems.

1.3.1 Standard Work

A lot of research has been done addressing these problems. One of the main efforts is the standardization of the new generation database systems based on the researches and development of a lot of different systems in the past decade. Nevertheless, current standards are only limited to several different data models, although they do absorb some new concepts from other data models. Along the relational and extended relational database systems, SQL3 is the on-going standard. Within object-oriented database systems, there are several different groups working on their standards, such as ANSI Object-Oriented Database Task Group (OODBTG, 1990), Object Management Group (OMG, 1990), and more recently Object Database Management Group (ODMG, 1993), etc. Standardization remains a long-term goal. At least in the near future, it may only reduce the problems, but will not eliminate it completely.

1.3.2 Front End Systems

Another approach is trying to provide some facilities for users and systems to use different platforms easily. Front end systems are one of them. Actually, a query facility that is provided to the users is only a front end of the underlying database system. The same underlying platform can support several different front end query languages. Even for relational database systems, because of SQL being developed by different vendors having some different flavors and extensions, a lot of front end systems have appeared on the market. For example, Sybase, as an underlying platform, may have an Oracle front end provided besides its own front end query languages. The data model of a front end query facility may also be different from the data model of the underlying database system, although usually they are the same. Some research has been done to build object-oriented front ends on relational database systems [13], and there are even some commercial systems like Persistence^{TM} [16] in the market. There is also some work trying to build relational front ends on object-oriented database systems[14].

1.3.3 Customized Query Languages

There are a lot application specific query languages, which have been defined and implemented on different database systems for some special applications. For example, TQuel is a relational query language that can deal with temporal predicates and it was developed on the relational database system Ingres [19]. Geo++ is a GIS system built on the extended relational database system Postgres, and it provides a spatial query language based on Postgres [15] [20]. DCQL, developed on the object-oriented database system Objectstore, is a CQL that can manipulate 3-D objects in a blocks world [11].

In relational database systems, the query languages usually do not support user defined functions or predicates. If users want to use some new predicates in the query language, like the temporal predicates in TQuel, the application programmers have to do a lot of work. Not only do they have to define and implement the new predicates, but also the need to redo a lot of query processing work in his application programs, even though most of the work, like parsing and optimization, is already done in the original query processor. This is because the interface between the relational database system and the user (including the application programmer) is quite simple and limited. In extended relational and object-oriented database systems, the query languages directly support user-defined functions and predicates. It is much easier to add new predicates into the query languages. For example, for the query language in Geo++ system, the developers only have to define and implement the new spatial predicates, and do not need to worry too much about how to integrate these new predicates into the original query language Postquel.

But if the CQL that the user wants to use has some different components or syntactic appearance other than the basic query languages, the developers may still have to redo a lot query processing work, like parsing, internal processing, overall optimization, etc. For example, in the development of DSQL, the developers had to define all the syntactic rules and do all the parsing work for the query, even though most of them are quite similar to the ordinary query languages. In current research and development, there is still no systematic approach for developing such kind of CQLs on new generation database systems.

1.4 Our Approach and Thesis Organization

Addressing the problems discussed above, in this thesis, we propose a systematic approach for query language specification and implementation which can help developers build different query front ends and Customized Query Languages for different applications on new generation database systems in the heterogeneous database environment.

Based on our studies of several popular new generation database systems and standards, we abstracted a lot of common components in the abstract level from the data models and query languages of these systems. These common components actually consist of an abstract common object model and query model. Based on these models, we also provide a default query language and some kind of mechanisms for modifications and extensions to the default query language, both semantically and syntactically. The description of the modifications and extensions consists of the special feature specifications of the target query languages users want to specify. According to these specifications, our system can automatically generate query processors for the target query languages.

For ease of use and understanding of our system, object-oriented methodology is used in our design of our abstract object model, default query language and the tools for modifications and extensions. This is to say that all the common components in the data model and default query language are represented by some kinds of meta-objects. The design of these common components embed the basic semantics of the object model and query model. Syntax of query languages based on these models are represented by the special kind of meta "constructors" of these meta objects. Query language specifiers can provide their own "constructors" for the meta-objects to override the default one in order to tailor the syntactic appearance and/or semantics of these components. New components and operations corresponding to some new components in the target query languages can also be defined through constructing some new meta objects and meta operations in a similar way to the new types and functions definition in C++. Since the whole query language specification is based on the pre-defined basic components, we can understand the syntax as well as the semantics of the target query language. Hence a query processor for the specified language can be generated automatically.

In our approach, language specifiers usually only have to specify their special features (comparing to the default one we provided) in their target languages. In this way, they actually can reuse both the specification and implementation of most of the common components in query languages, and their work of building customized query languages can be dramatically reduced.

Another more important advantage of our approach is that in multidatabase environment, our high level specification of a query language provides a high independence for the implementation of the query processor for this query language from the underlying platforms. Actually, language specifiers can concentrate on language design itself, do not have to worry about how to implement it on different database systems. Because our system can generate the query processor automatically for them.

In the following chapters, Chapter 2 will give some more general overview of our approach. Chapter 3 will concentrate on the common components in the Object Description Languages (ODL) and the object-oriented specification of these components. Chapter 4 is the counterpart of Chapter 3 for the Object Query Languages (OQL). In these two chapters, we also use a lot of language examples to illustrate the functionality and flexibility of our language specification language. Implementation algorithms are described in Chapter 5, and some testing examples are also discussed in this chapter. Chapter 6 discusses our conclusions, and some further work is also suggested.

Chapter 2

Design Overview

In this chapter, we will give a general overview of our approach and explain the basic underlying philosophy of our system design. This mainly includes the following points:

1. **Front End Approach:** We separate query processors from the underlying platform, so that different query front ends can be built on one underlying platform.

2. **Abstract Object Model and Query Language:** We abstracted a lot of basic components from the object models and query languages of different new generation database systems, and these abstract common components formed an abstract object model and query language.

3. **Object-oriented Design of the Abstract Object Model and Query Language:** All the components in the object model and query language are represented by meta objects and meta-types, for which meta attributes and meta operations are defined. Language specifiers can customize the object model

and query language into the query languages they needed by providing their own "constructors" for these meta objects. These "constructors" actually provide some kind of mapping of the data models and query languages from the specified query language to the abstract ones we provide in our system.

4. **Constructing New Components for CQL:** For customized query languages, there may be some special components which are not easy to be mapped directly onto the components of our abstract query language. Our system also provide some mechanisms to the language specifiers to construct these special components based on our abstract components in the abstract object model and query language. These special component constructions actually serve as some complex mapping from these language specific components to some ordinary components in our abstract query languages.

5. **Automatic Generation of Query Processor for the Specified Query Languages:** According to the specification of the query language based on our system, our system can automatically generate a YACC/LEX program for the specification and further generate a query processor for the specified query language.

In the following subsections, we explain these points one by one.

2.1 Front End Approach

Query languages are interfaces of data definition and data manipulation provided by database systems. They can be treated as front ends of the underlying database platforms. A query language is usually based on a certain data model in which

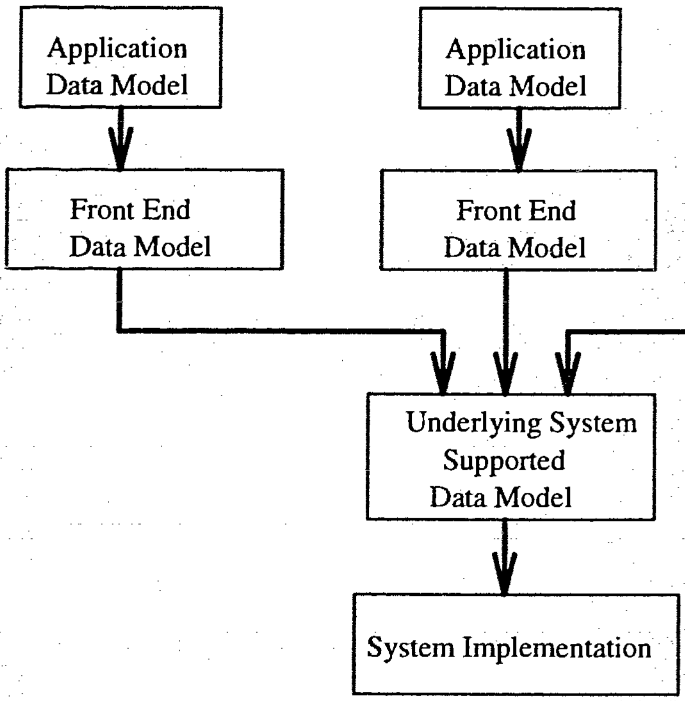

Figure 2.1: Data Model Mapping From Front End to Underlying Platform

the users will define and manipulate their application data. The data model of the front end should be close to that of the application so that the mapping between the application data model to the front end data model can be natural and efficient. But the data model of the front end need not be the same as the one of the underlying platform, as long as we can have an automatic and efficient way to map the front end data model to that of the underlying database system (see figure 2.1). So the same underlying database system may support different front ends (figure 2.2), and one query language (including some CQL) can be implemented on different underlying platforms (figure 2.3).

Our purpose of the separation of front ends from underlying platforms is to try to provide a maximum insulation between the front end of the database systems that the users use and the underlying platforms that may be different in the network. This kind

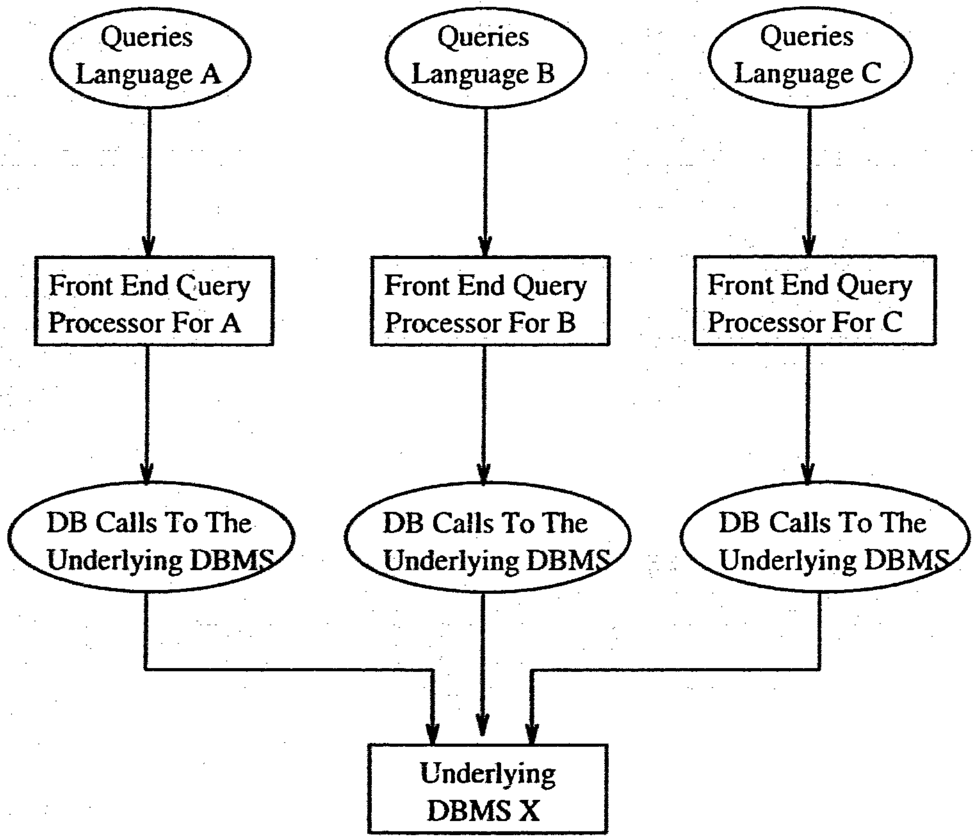
Queries Language A
Queries Language B
Queries Language C
Front End Query Processor For A
Front End Query Processor For B
Front End Query Processor For C
DB Calls To The Underlying DBMS
DB Calls To The Underlying DBMS
DB Calls To The Underlying DBMS
Underlying DBMS X

Figure 2.2: Different Front Ends on One Underlying Platform

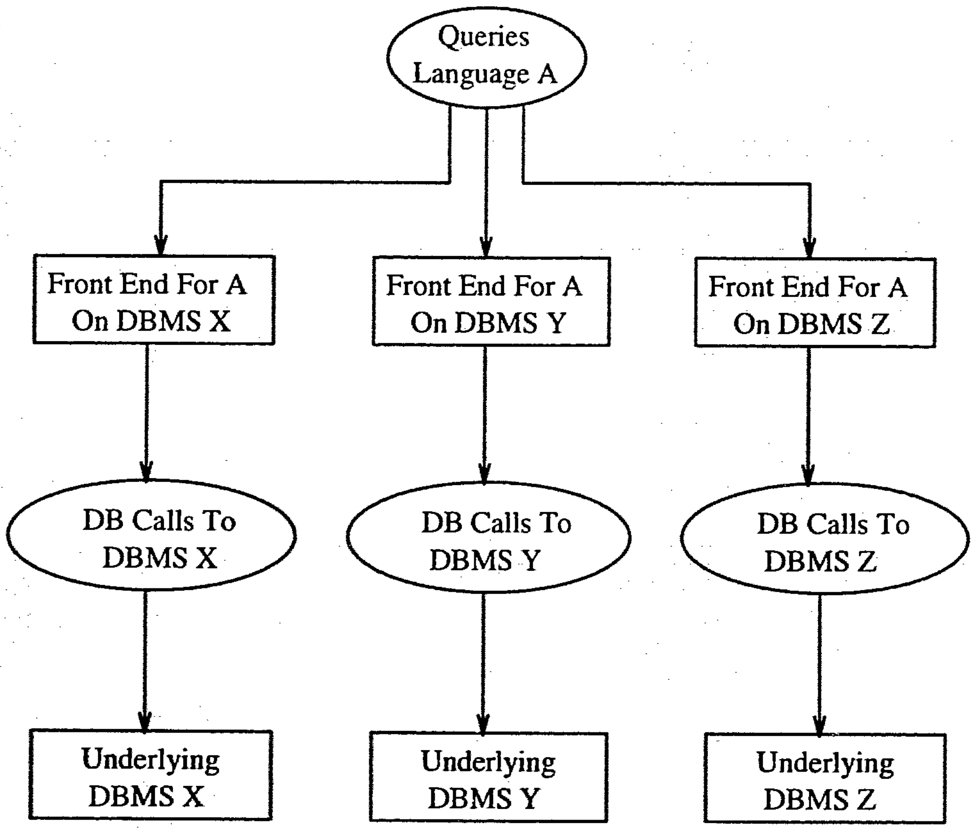

Figure 2.3: One Query Language Implemented on Different Underlying Platforms

of insulation first can be very useful for the end users, because they only need to learn one kind of front end query facility and phrase their queries in this query language, even though they are actually using different platforms. For example, suppose a user is familiar with O2 query language, and now he has to use a database system with the underlying system of Objectstore (which currently does not provide any stand-alone query language). With an O2 front end built on Objectstore, the user can use Objectstore as if it were O2. Secondly, it is also good for the application developers: when they develop a certain application specific query language (ASQL or CQL), they do not have to worry about the underlying platforms. Instead they can concentrate on the front end designing itself. Actually, a CQL can also be treated as a kind of front end of the underlying platform. A query based on a CQL can be mapped onto a query facility (stand-alone or embedded query languages with some help of other components in the host language) of the underlying platform directly. We will explain more about this later.

2.2 Abstract Object Model and Query Language

The problem of whether it is easy to support different query front ends on one underlying platform depends on how different the front end object models/query languages are from those of the underlying systems. Although there are a lot of different new generation database systems, and different systems have different data models and query languages, after our studies of several popular new generation database systems and standards, we found a lot of commonalities in most of the systems. The reason for this phenomenon is that the main goals and desired capabilities of these systems are quite similar. As summarized in [4], "The new data models generally provide similar functionality". Especially in recent years, all the data models incorporate a lot of object-oriented concepts into them to deal with the problems of complex data management, and they began to merge into one data model. "The important observation to make from this confusion of data models is that many data models are slowly converging on a single data model that has a combination of features." So the difference between the data models is superficial compared to the small difference of the underlying semantics, and "The data model is most useful as a way to distinguish the genealogy of a system;"[4]. Different query languages based on these data models may have different flavors or dialects, but the differences are more in syntactic aspects than semantic ones.

From our studies, we abstracted a lot of common features from the data models and query languages of different new generation database systems and standards. These common features, which can be divided into two main parts, META-ODL and META-OQL (corresponding to Object Description Languages and Object Query Languages), capture the basic semantics of the data models and query languages of different new generation database systems. In a sense, all these common components consist of a core abstract object model and a kernel query language. But the core abstract object model and the kernel query language are at an abstract level. Because the common features are abstract structures which describe not only the basic common semantics, but also the special features of different systems. Based on the abstract object model and query language, we can describe various object models and query languages by providing a specification of the special features of that query language and object model.

2.3 Object-Oriented Design of the Abstract Object Model and Query Language

There are a lot of basic common components in the abstract object model and query language. Among these components, there are a lot of different constraints and relationships. For the ease of use and understanding of the general framework for query language specification, we used object-oriented methodology in designing and presenting the whole model. That is to say, all the components in the model and query language are abstracted into some kind of meta-objects and meta-types. All these meta-objects and meta-types have their own meta attributes. And we also define some special meta-operations on them.

Different concrete query languages based on the object-oriented data model can be specified by customizing some of the meta-objects and meta-types. The syntactic appearance of query languages will be presented as a kind of special operations (which are called meta-operations) defined on the meta-objects. All these operations have default formats derived from the corresponding components in the popular query languages, and these default formats actually consist of our default query language for the new generation database systems. If any actual query language is only different in some features, the language specifiers will only have to specify the differing parts by providing their own meta-operations to override the default ones. In this way, the language specifiers will actually reuse a lot of work that is common to most of the query languages.

Since we understand the underlying semantics of all the meta-objects and meta-types in our abstract data model, the semantics of any query language specification

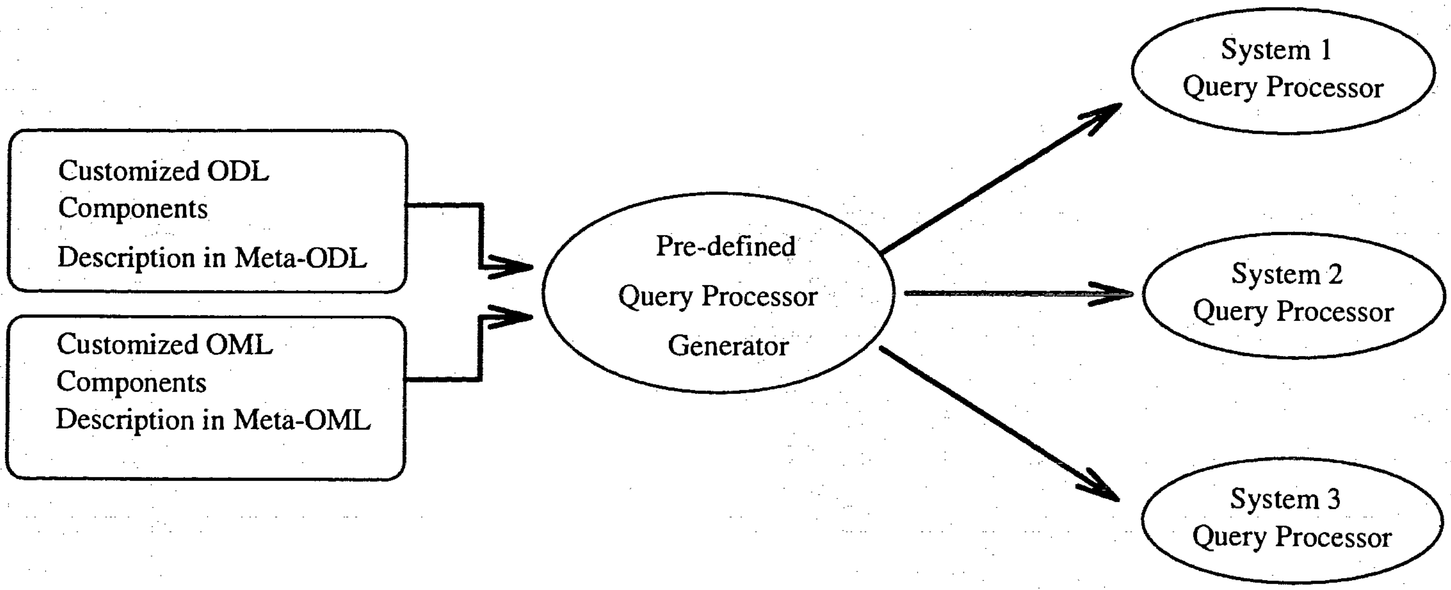
Customized ODL Components Description in Meta-ODL
Customized OML Components Description in Meta-OML
Pre-defined Query Processor Generator
System 1 Query Processor
System 2 Query Processor
System 3 Query Processor

Figure 2.4: Our Approach for Query Processor Development

that is based on the meta-objects and meta-types will be easily understood by our system. So most of the syntactic appearance specification (in the form of overriding of some meta operations) will also be some kind of semantic specification. Language specifiers will only provide some high-level semantic actions for some special mapping of some components of object models and query languages if necessary, and our system will be able to generate a query processor according to the specification. In some sense, the language specification will also be the implementation of itself. This will dramatically reduce the amount of work for the language implementation.The general picture of this process is illustrated by the figure 2.4.

From the figure, we can also see that we can actually generate the query processors for the language specification on different platforms. This is because it is not difficult to map our abstract object model and query language to the object models and query languages of those actual systems. In this thesis, we will not elaborate this in detail. Instead, we will use some examples to illustrate how we can map the common components of our model onto those of the Objectstore system, since the mapping onto other OODBMS is quite similar to the mapping we described for Objectstore.

2.4 Constructing New Components for CQL

As we stated before, a customized query language usually has some special components in the query language. For example in TQuel, they provide temporal predicates; in Geo++, they provide spatial predicates; in DSQL they even used different vocabularies of the commands, i.e. provide some other formats (which are more natural for the application) of the data manipulation components, such as move, rotate, etc. Some of the special components can be easily fit into the ordinary general query facilities provided by the underlying new generation database systems. But some of them may not. If this is the case, then the developers of the CQL may have to redo a lot of work even for some ordinary components in the query languages, like parsing, and more difficult, the internal processing and overall optimization. This is because, although this work has been done for the general query facilities, it is not directly available to the developers. This is one of the main reasons why currently there is no systematic way to specify and implement customized query languages.

In our system, since we treat a CQL as a front end of the underlying platform, we also provide some facilities for CQL specification. Usually, the developers of a CQL will only have to specify the special components that are not easy to fit into the general query facilities. This is done through the construction of special components based on our abstract object model. Since the construction of the special components is quite similar to the new type or new method definitions in ordinary object-oriented programming languages, like C++, it will not be difficult for ordinary developers. And because the constructions of the special components will actually be based on the pre-defined basic components, they will define the relationship between the special components and the pre-built components in our system. This will reveal the relationship of the CQL and the general query facilities, and the system will be able to automatically generate a query processor for the CQL.

2.5 Implementation Consideration: Automatic Generation of Query Processor

Generally, if we want to build a query language on a new generation database system, first we have to formally specify the language in some format. Usually, an ordinary formal language is specified in two parts: syntax and semantics. The syntax is specified in the general BNF or its variants' format, and the semantics is described in the semantic actions (described in a programming language) hooked to the BNF rules. There are some compiler generator tools, like YACC which will help the implementation of the language. But it still involves a lot of work, from the parsing to the internal processing and optimization. (see figure 2.5)

In our approach, as discussed above, it seems that we use a rather different way to tackle this problem. But actually, there is a close relationship between these two different approaches. From the implementation point of view, we first map the specification based on our framework onto the general approach. That is to say, we will generate the BNF syntax rules and also semantic actions hooked to these rules from the specification given by the language specifiers. More specifically here, we generate a YACC program for the specification. Then from the YACC program, we

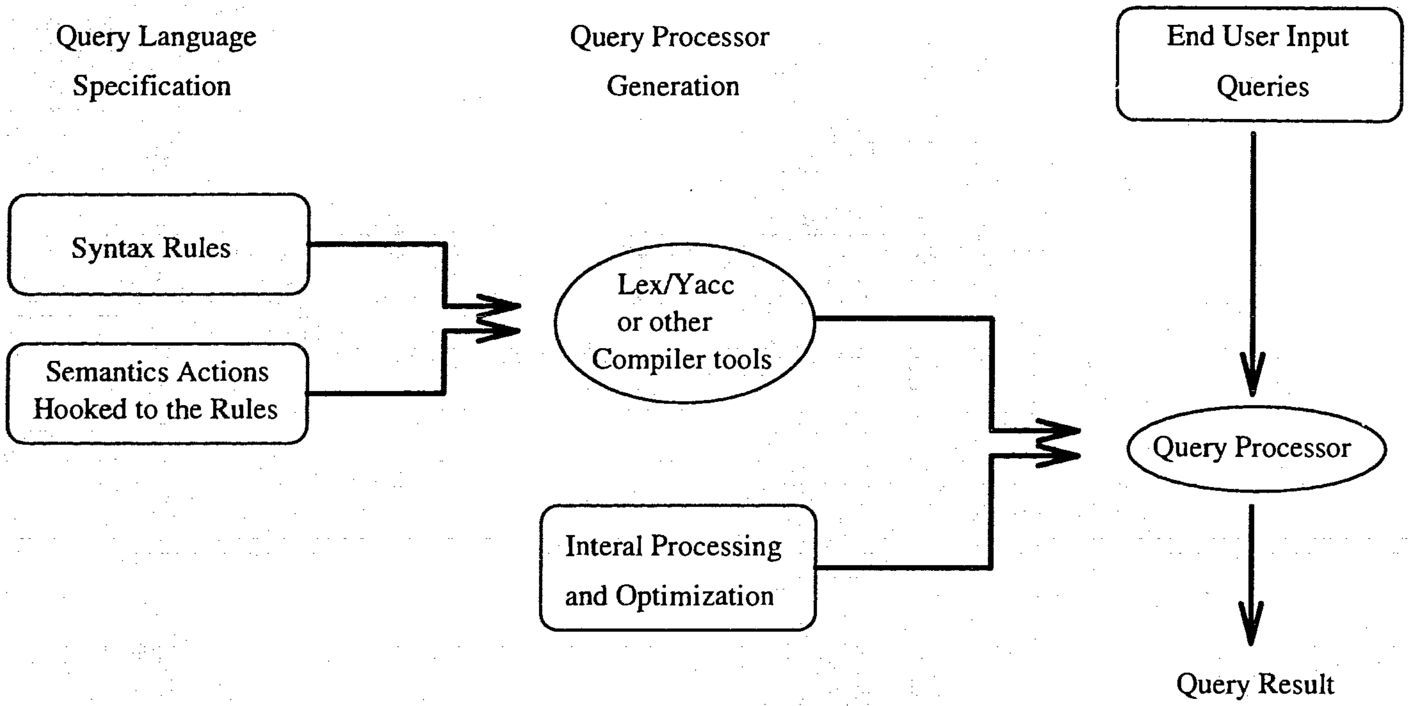
Query Language Specification
Query Processor Generation
End User Input Queries
Syntax Rules
Semantics Actions Hooked to the Rules
Lex/Yacc or other Compiler tools
Query Processor
Interal Processing and Optimization
Query Result

Figure 2.5: General Approach for Query Processor Development

can further generate a query processor for the specified query language.

To illustrate the relationship between our framework and the general formal language specification, and also explain some basic concepts that we will use in the following chapters, in the next section, we will present an overview of general formal language specification and how our framework deals with all these aspects of the formal specification of query languages for new generation database systems.

2.6 Language Specification Overview

The specification of a linear (or one dimensional, not two dimensional like QBE) language should include the following aspects, from low level to high level:

1. **Lexical Elements and Lexical Rules:** Which include the specification of character set (including some special characters, and some other features, like case sensitive or not, etc), tokens, separators and literals; For the tokens, separators and literals, some lexical rules are associated with them, describing how they are made from the characters in the character set.

2. **Non-terminal Syntactic Symbols:** Which represents the basic concepts in the specified language. In ordinary programming language specifications, this includes < Program >, < Function >, < Statement >, etc; and in SQL, < Table Definition >, < Column Definition >, < Query Expression > are some typical syntactic symbols.

3. **Syntactic Rules:** Which describe the appearances of the syntactic symbols, i.e. how the higher level concepts are made from the low level concepts in the specified language.

4. **Language Semantics:** While the above three aspects only describe how the language looks like, the language semantics will describe what is the actual meaning of the language. Although there are several formal methodologies of defining the semantics of formal languages, such as Operational Semantics, Denotational Semantics, Axiom Semantics, etc, usually it is not easy to do that. In practice, nature languages are used in the language documents and semantic actions described in some programming languages are used in some compiler generators, like YACC.

In our framework, we need to provide the corresponding facilities to the language specifiers to describe the above aspects of the special formal languages: query languages for object-oriented database systems.

2.6.1 Lexical Elements and Lexical Rules

Currently we assume our character set is the ordinary ASCII set, including some of the non-visible characters, like '\n', '\t', etc. Although, other kind of character sets can also be specified easily, we will not consider this in our thesis.

Tokens and separators are the basic lexical units in query languages. Tokens include keywords, regular identifiers, etc; separators include space, newline, comments.

Tokens and separators are usually associated with some kind of lexical rules. In our system, we have some default rules for some default tokens and separators. Language specifiers can use the default tokens and separators, change the default lexical rules of these default lexical units, or define their own tokens and separators. Tokens and separators declarations are the first part of a language specification, and in our system, all of these can be done by the token declaration part at the beginning of a language specification. For example, we can declare that

```
TOKEN IDENTIFIER  {letter}{letter-or-digit}*
```

which means that IDENTIFIER is a TOKEN and it consists of a string of letters or digits in our character set. The last part of the above declaration is in regular expression format and means that the string corresponding to IDENTIFIER must start with a letter. If the language specifiers will not give the regular expressions for the token here, we assume that he will provide the information of how the token is composed in another place. Having been declared, the TOKEN can be used in the specification. Generally, these kinds of lexical elements and lexical rules are dealt with in a way similar to YACC and LEX.

Literals include the literals of the built-in types and literals of the constructed types defined by the users. A numeric literal is a literal of built-in types (INTEGER, FLOAT, etc). A string literal is usually also a literal of built-in types(CHAR, VARCHAR, etc), which usually consists of a string body enclosed in double or single quotes. These kinds of simple literals are also described by lexical rules. For the constructed types, we should provide some more complex rules, usually in the format of syntax rules, to construct the literals of these types. We will discuss this in the following chapters.

2.6.2 Non-terminal Syntactic Symbols

Non-terminal syntactic symbols represent the basic concepts in the specified language. In query languages conforming to the object models, we should already know most of the basic concepts in them, according to our study of object models and query languages. Here we use OO methodology to analyze and represent object models and query languages. We can abstract the widely accepted concepts into a set of types and objects. These are the *meta-types* and *meta-objects.*

The meta-types and objects are the center of our system. Most of the non-terminal syntactic symbols in the query languages that will be specified are objects or components of objects in our system. When we describe any meta-types or objects, we list all the components of the types. For each component, we define an attribute name and a domain of this attribute. The domains of the attributes in our meta objects can be TOKENs, other meta-types or meta constructed types (like list) defined in our model. If the domain of an attribute is a TOKEN, then the language specifier can redefine the domain as another TOKEN defined before as a sub-type of TOKEN (for example IDENTIFIER defined above). The definition of the (meta-)attributes in the meta objects using (meta-)domain actually reveals some aspects of the relationships between the basic concepts in the specified languages. We will explain this further with some examples.

2.6.3 Syntactic Rules

Just like ordinary types and objects, all the meta-types and objects also have (meta-)operations defined on them. These operations are in the special format, i.e. the parameters of the operations are represented by lists of *"Comma Expression"*. Constructors of the meta-types are one kind of special operations which define how the meta-objects are constructed from their components. And these just represent the syntactic rules of how the higher level symbols are made from the lower level symbols. In Chapter 3, we have a lot of examples to illustrate how the meta operations and the parameters to the meta operations look like. A rather formal description of them will be presented in chapter 5. The algorithms of how syntactic rules in BNF format can be generated from the operations defined on the meta-objects will also be described there.

2.6.4 Language Semantics

Most partS of the semantics of the specified languages are actually predefined because of the predefined OO data model. The predefined meta-types, meta objects and their components actually represent some semantics, and wherever they appear in the target language specification, the system understands what they stand for and corresponding semantic actions will be performed in the generated query processor. This is the same for the syntactic rules represented by the meta-operations predefined on the meta types and meta objects. We will explain the corresponding semantics of the components of our meta languages (i.e. in some sense, we provide some operational semantics for the specified languages) in Chapter 3 and Chapter 4.

Language specifiers can also provide their own semantic actions in some special cases. This is because for some languages, the concepts in their object model may not be able to be mapped directly onto the concepts in the object model we provide here. Then the language specifiers must provide some more semantic actions for some components, specifying the mapping methods from the concepts in the specified language to the components of our abstract object model and query language. These semantic actions are usually attached to the comma expressions or meta operations, reference some corresponding meta objects and in the format of C++ statement. We will explain this more with some examples in the following chapters.

Chapter 3

Abstract Object Model and ODL

3.1 Object-Oriented Data Model Overview

The Data model is the basis of any database management system. It is a logical framework in which the real-world data will be represented by users, and it also defines the framework of how a system will manage the application data. All interfaces between a system and users, such as data definition and data manipulation language, should conform to this model and are only some kinds of concrete forms of the data model.

There is a lot of research on object-oriented data models in recent years, and people begin to agree on most of the basic concepts, i.e. some "core concepts" of the data models. The agreement on most of the basic concepts of object-oriented data models provided the basis for the work of many commercial products of object-oriented database system and standardization efforts of these kinds of systems. From our studies of several object-oriented systems such as Objectstore, O2, Orion, some extended relational database systems such as Postgres, and especially some of the popular standards in these areas, such as SQL3, OMG, ODMG-93, we found that there are a lot of common components in the data models and query languages at an abstract level. These abstract common components formed an abstract object model and an abstract query language based on this model. In this chapter, we will analysis and define these common components in our object model and object definition language.

For the purposes of this thesis, we will not describe a complete object model here. But rather we only include some of the basic aspects of building general stand-alone query facilities, i.e. basic components in Object Definition Language (without schema evolution, version management, etc) and Object Query Language (query statements themselves, without transactions management, etc). This chapter will describe ODL part. The part for OQL will be left for the next chapter.

Generally, in an object-oriented data model, *object* is the basic unit of modeling. Objects are classified into *types*, and *types* are organized into type hierarchy under the relationship of super/sub-type. As the basic unit of modeling the application world, an *object* is characterized by a set of states and behaviors. *Objects* of the same *type* have the same behaviors and the same set of state ranges, and these objects are called instances of this type. Along the line of *object* and *type*, there are some other basic components in an object model. For example, to represent the states of an object, we have *attributes* of the object, *relationships* between objects; we have *operations* to model the behaviors of objects. Between sub-type and super type, we have *inheritance* to represent their relationship, etc. All these components have their own characteristics. For example, an *object* has an *object identifier, type*, and *lifetime*, etc. It also has a set of operations, such as *create, delete*, etc. All these represent the basic concepts in an object model.

In the following sections, we will analyze the basic components abstracted from these basic concepts of general object models. Usually, in our abstract model, each abstract component is represented by a meta type which corresponds to a component in some concrete object models and query languages. For example, we have an abstract meta type OBJECT which corresponds to the basic concept of *object* in ordinary OODBMS's, and we also have an abstract meta type TYPE which corresponds to the concepts of *type* or part of *class* in different systems. For each of the basic abstract components, we will first give a general discussion about this component, describe the basic functionalities of this abstraction, and its basic characteristics and operations that we provide to query language specifiers. Then we define this component as a meta type in an object-oriented manner using our *meta language*. [1] Finally, when necessary, we use some examples to illustrate how different modifications and extensions can be done to this component of our object model and the part of the corresponding ODL.

3.2 OBJECT

Object is the basic unit of application modeling and also the basic unit of data management. Although in most systems, most of the characteristics of an object are not directly visible to end users through the Object Definition language, and most of the

[1]The format of a definition in our meta language is quite similar to the format of a class definition and function prototype definition in C++. The only difference is that we use our *meta types* and *meta type constructors* instead of C++ types and constructors when we define the *meta attributes* of the defined components, and use *comma expressions* as the parameters when we define all the meta operations. Hence, we will not give a complete formal description of our meta language. Instead we will just explain the special features of our meta language along with some examples in the following description.

operations on objects are only part of the Object Query Language, which we will discuss in the next chapter, some of the basic semantics of object are important for the following discussion of other components in ODL. Hence in this section we introduce our meta type OBJECT as the abstraction of the concept of object. The basic attributes and operations of OBJECT are based on the basic attributes and operations of an object in most OODBMS's.

3.2.1 Basic Attributes

An object in most systems has the following attributes, and in our abstract object model and meta system we also support these basic semantic of objects:

- **Object ID (OID):** Each object has its own unique object identity, which is usually called as an object identifier (OID). The OID of an object remains the same even if the states of this object may change and an old OID will not be used even if the associated object has been deleted from the system. In some systems, the user can use OID function to get the OID of any object and then the OID can be used to pin-point the object.

  Usually, OID is not directly visible to end users, and the pattern of an OID is also implementation dependent. We will not directly support any special semantics of OID in our system.

- **Type:** Right now, in all the systems we have studied, each object has one and only one the most specific type(MST, which means that if the object belongs to any type, then the MST of this object must be a sub-type of this type). When an object is created, usually the type is also specified. In some systems, some kind operations are also provided to the end users to get the MST of any object.

- **Lifetime:** Most systems assume that the lifetime of an object is orthogonal to its type. It is either transient, which means that the object will not exist any more after the process that creates the object finishes; or persistent, which means that the object will outlive the process and usually will be stored in a non-volatile storage. Lifetime of an object is usually indicated when the object is created.

- **Names:** In some systems, user can define names for objects. A name is a kind of meaningful identity of the named object so that after named, the object can be referred later by this name. An object can have several different names, but it can have only one OID.

  A name can be transient, like an ordinary variable name (but the name of an object can not be given to another object while a variable name can), or persistent which means that we can use the same name to refer an object in different processes. Right now most systems only support one single name space within a database for the persistent names. This is to say, the same name must refer to the same object in the whole database. So in our meta system, we only directly support a single persistent name space.

  Syntactically, a name of an object is usually a string with a certain set of lexical rules, or we say that it is a certain kind of TOKEN in our system. Usually we call this kind of TOKENs as IDENTIFIERs which are a kind of sub-type of TOKEN.

3.2.2 Basic Operations

In our meta system and abstract object model, we have the following default built-in operations for an object, reflecting the basic operations for objects in an ordinary OODBMS:

- **create:** This operation will create an object according to the specified type and lifetime (which are the two main parameters to the object creation operation). In our meta system, there is a default built-in create operation which will be mapped to the underlying database functions that will do the basic work for an object creation, such as OID generation, storage allocation, etc. In many systems, different types can have their different instance creation operations, i.e "constructors". These different constructors may override the default one and also initialize some attributes of the object. But eventually they will call the basic object creation operation. Special instance creation operations for some types will be discussed later.

- **delete:** This is the counter operation of create operation. Although we have a default built-in delete operation for all the objects, different types may define their different delete operations which we call "destructors'. Usually these "destructors" will override the default one, but again they will also call the basic object delete operations eventually. Delete operations only need the OID or name of the object that will be deleted

- **equality:** This is the function or operator which is used to test whether two parameters refer to the same object, i.e. the OIDs of the two objects are the same or not.

There are several different equality semantics in different object oriented data models. In our thesis, for simplicity, we only consider this kind of *OID equality*. Other kind of equality semantics can also be analyzed in a similar way.

- **OID-function:** This is the function that returns the OID of an object. In some systems, the OID is not visible to users and there may be no such a kind of function provided to end users.

- **MST-function:** This is the function which returns the most specific type of an object. This operation has to retrieve the meta knowledge, i.e. the schema of the database. In object-oriented database systems, this kind of meta knowledge is very useful to the users.

3.2.3 Specification and Explanation

According to the above discussion, a meta type OBJECT can be defined in our meta system as follows:

```
OBJECT {
  TYPE                   Type;
  LIFETIME               LifeTime;
  list(IDENTIFIER)       Names;
  // default built-in operations on OBJECT.
  create("create", LifeTime, Names.ele, "of",  Type);
  delete("delete", Names.ele);
  OID oid("oid", "(", Names.ele, ")");
  TYPE.Name MST( "type", "(", Names.ele, ")");
}
ENUM LIFETIME{
  KEYWORD               Persistent;
  KEYWORD               Transient;
  // default built-in specification of KEYWORD.
  Persistence("persistent");
  Transient("transient");
  // defining the default for a meta enumeration type.
  Default(Transient);
}
```

As the first component specified in our meta system, there are a lot of places that should be explained. But here we only explain the following basic points about the specification. More explanations will be presented in our later examples:

1. The meta type definition consists of a list of meta attribute definitions and a list of meta operation definitions which are the abstraction of the common attributes and operations of *objects* in different OODBMS's. Since OID is not directly visible to the users, we only specified the other three attributes of the meta type OBJECT.

2. The definition of each meta attribute consists of an attribute name in the right side and a meta domain of this attribute in the left side. The definition of a meta attribute usually has both semantic and syntactic meanings, which are mainly dependent on the meta domain of this meta attribute. There are several different kinds of meta domains, including TOKEN, another meta type and constructed meta type, etc. We will explain these with some examples later.

3. The meta attribute `Type` represents the (most specific) type of the object, and its meta domain is `TYPE` which represents all the possible type specifications (both semantics and syntax) in the specified language collectively and we will explain this concept later.

4. The meta attribute `LifeTime` represents the lifetime of the object, and its meta domain is a meta enumeration type `LIFETIME` which is also defined above. The meaning of this definition is obvious, it represents what kind of lifetime an object can have. More detailed discussion about meta enumeration type definition will be presented later.

5. The meta domain of `Names` is a meta parameterized type `list(IDENTIFIER)` which means that object can have a list of names, each of which is an IDENTIFIER. IDENTIFIER as we declared in Chapter 2, represents a kind of token with a certain set of lexical rules. Meta type constructor `list` and several of its operators (such as `.ele` used above) will be explained later, too.

6. We specified some default built-in operations for the meta type OBJECT. Semantically, they specify what kind of operations are provided for objects in the specified languages, and what kind of parameters are needed for these operations. Syntactically, they also represent what these operations look like in the specified language. So these specifications actually connect the syntactic appearances of the operations with the underlying semantics in the specified language. As we can see from the above example, the format of the meta operation specification is quite similar to an ordinary C++ function signature specification. The only difference is that each of the parameter for the operations is a *Comma Expression* as we mentioned in Chapter 2. We have several different kinds

of *Comma Expressions* which have different semantic and syntactic meanings. Here we have three kinds of *Comma Expressions*. In the meta operation `create` (which is the name of the meta operation), `"create"`, `"as"` are called *String Comma Expressions* which mean that they will appear themselves in the object create statement in the specified (here is the default) query language. `Type`, `LifeTime` are the meta attributes of the OBJECT, (and `Names` in `Names.ele` is also an attribute of OBJECT, `.ele` is an abbreviation of *element*, representing the element in the list), they represent the corresponding semantic and syntactic meanings of these meta attributes as we explained above. So we know that, for example, in this object creation statement, the last string should conform to the lexical rules of `Names.ele` (i.e. IDENTIFIER), and should be saved as the name of the created object.

According to the above specification, we know that there is an object create statement in the default query language. The information needed by this statement is the *life time*, *type* and the name of the created object. And also, we know that the format of the create statement should look like:[2]

```
create [persistent | transient] <obj-name> of <type>
```

where the `<obj-name>` is an IDENTIFIER, and the format of `<type>` will be described later on. The specification of other meta operations are similar to this create statement.

[2]We just use some simple forms which is similar to BNF rules to illustrate the format of the statement. In chapter 5, we will describe how the syntax rules in YACC BNF format can be generated from our specification.

3.2.4 Examples of Modifications and Extensions

Similar to ordinary Object-Oriented (OO) programming languages, such as C++, language specifiers can override the specification of the default built-in operations by providing their own specification for these operations. The overriding of a built-in operation means that a similar operation does exist in the specified language, but the format (hence the syntax and also semantics) of the operation may be different from the default one. Here we use a small example to show how some simple modifications can be done to customize the default query language. For example, language specifiers can provide their own object create operation to override the above default one in the following format:

```
OBJECT::create(LifeTime, Type, Names.ele);
```

where `OBJECT::` specifies the scope of the meta operation `create` is within OBJECT. According to this meta operation, in the specified query language, objects are created as follows:

```
[persistent | transient] <type> <obj-name>;
```

which is similar to the format in the Objectstore and O_2 system.

Some other variations can be done to the above specification, and a lot of more will be done through the variations of the specification of the type system for the specified language, which will be described later in the corresponding sections. But no matter what kind of specification it may be, we know that in the position of `Names.ele`, the string is served as the name of the created object, in the position of Type, the part of specification will be processed as type information of this object, etc. And after all, all these information will be passed to the creation operation of an object in the underlying database system because we know that this is an object create statement from the predefined meta operation name `create`. So we can see that the specification serves as both syntax format and semantics.

3.3 TYPE

Although *object* is the basic unit of modeling in object models, objects are usually grouped into types, and type is the basic way of defining the states and behaviors of objects. The type system is one of the most important part of any object-oriented database system. Usually, there are some pre-defined basic types and type constructors in an object-oriented database system. And the system also provides some kind of mechanisms for new type definition. Schema definition through type definition is one of the central parts of ordinary Object Definition Language. Here we provide the meta type TYPE to capture this basic concept, and provide it to language specifiers to specify those features of new type definitions in their query languages. The corresponding meta types for describing pre-defined types in the type system will be described later in this chapter.

3.3.1 General Discussion

In the object-oriented field, including many object-oriented database systems and object-oriented programming languages, sometimes another concept *class* is used instead of type. In different systems, the concept *class* may have different meanings. First, a class defines the *intent* of a type, i.e. it defines the characteristics, such as states and behaviors of the objects of this type. This is the basic meaning of a type. Secondly, it may also define the *extent* of the type, i.e. the set of all the instances of this type. This is not an essential aspect of a type and may not be defined in the type system in some systems. For example in most OO programming languages, they do not define nor maintain the extent of a type. Although in ordinary OODBMS's, it is important to maintain the extent of a type, we believe that it is better to separate the *extent* specification from the *intent* specification of a type. So in our system, a type only defines the intent. We will provide another concept, i.e. *persistent collection*, to specify an extent of a type, so we can even define more than one extent of a given type. This is quite similar to the idea in SQL3, which can define ADTs (types) and more than one TABLE (persistent collections) for a given ADT. And a TABLE of an ADT is actually an *automatically maintained* persistent collection of this type.

In some OODBMS's, such as Objectstore, they can define some other attributes or constraints on a type, for example *keys*. But this is because they mix the extent of a type with the intent of the type. A *key* is only meaningful when there is a collection of the instance of that type maintained. Otherwise the system can not use the key to distinguish or retrieve the objects. So in our type concept, we do not have these kinds of attributes or constraints.

3.3.2 Basic Attributes

The following is an outline of the basic attributes of the meta type TYPE abstracted from various systems. They actually represent the basic characteristics of a type definition that different object oriented query languages may have.

- **Name:** Each type has a name which is the identifier of this type. It should be unique within a certain name space. Right now, all the systems we have studied use a single name space within each database. So in our meta system, we also take this assumption. Similar to the names of objects, names of types are also a kind of TOKEN with a certain set of lexical rules which may not be the same as those for object names. For example, we may use TYPE-IDENTIFIER to represent this kind of TOKENs.

- **Inheritance:** Types in an object-oriented system are organized into type hierarchy under the relationship of sub-type and super-type. A type can specify its super-types. This implies that it may inherit some characteristics(i.e. attributes, relationships and operations) from its super-types. A type may have more than one most specific super-types in our system. All these kinds of features about type definition will be described in Inheritance description of this type. The details about inheritance description are abstracted into the meta type INHERITANCE, which will be discussed later in this chapter.

- **Attributes:** Usually a type has to define the states of its instances, and the states of an instance include a list of attributes. Attributes in our meta type TYPE is used to capture this kind of features of type definition. The detail of how to define each attribute is abstracted into the meta type ATTRIBUTE, which will be discussed later.

- **Relationships:** Relationships are also part of the state definition for the instances of a type. They are used to represent the binary relationships between the defined type and another type. Relationship is one of the mechanisms of maintaining referential integrity. Usually, a relationship is decomposed into two parts for the two involved types. Each part is called *traversal path* in our system (and some other systems, such as ODMG-93 etc). So actually, in a type definition, we only define one part of a relationship, i.e. the *traversal path* in this type, instead of the whole relationship. A traversal path can be used in navigational retrieval from the instance of one type to the instances of the other type. More discussion about that is under RELATIONSHIP component of our abstract object model.

- **Operation:** Object behaviors are represented by a set of operations. An operation specification usually can be divided into two parts: its signature, which specifies the name of the operation, the parameter list and the type of the return value; and its implementation. Usually these two parts can be defined separately, and even in different (programming or query) languages. More discussion about operations is under OPERATION component of our data model.

In some systems, the attributes and relationships are collectively called *properties* of a type. Properties together with operations oı a type are called *characteristics* of this type. In our system, we use meta types PROPERTIES and CHARACTERISTICS to represent these two concepts respectively, and they will be discussed later in this chapter.

3.3.3 Basic Operations

The following operations are some default built-in operations on TYPE. They are abstracted from the basic operations or functions on types provided to end users by ordinary OODBMS's. Through these operations, language specifiers can specify how these operations can be provided to end users in the query languages they want to specify.

- **create:** Type creation forms a new type with a list of characteristics: i.e a list of attributes, relationships and operations (signatures only) of the type. Usually, a name is also provided when a user defined type is created.
- **delete:** This operation drops the type from the system. Usually, only the type name is needed for this operation.
- **supertype:** This function returns all the immediate super-types of the type. Usually, only the type name is needed for this operation.

Here, we do not include some advanced functions or operations, such as schema evolution operations, etc. But these advanced functions can be analyzed in a similar way as those above.

3.3.4 Specification and Explanation

According to our previous analysis, the meta type TYPE in our meta system is defined as follows:

```
TYPE {
    TOKEN                                              Name;
    INHERITANCE                                        Inheritance;
    list(CHARACTERISTICS)                              Characteristics;
    // default TOKEN definition for Name;
    Name(TYPE-IDENTIFIER);
    create("create", "type", Name, [Inheritance], "{",
             Characteristics, "}");
    delete("delete", Name);
    LIST(TYPE.Name) supertype("supertype", "(", Name, ")");
    // default separator of the list:
    Characteristics.sep = ";" ;
  }
```

There are several points we should explain for the above definition of the meta type TYPE:

1. The meta domain for the meta attribute `Name` is TOKEN, which means that the name of a type should be a token terminator in the specified language. In our system, for all the meta attributes whose domains are TOKENs, language specifiers can override this by providing their own special TOKENs for this attribute. For example, we provide TYPE-IDENTIFIER as the default special TOKEN domain for the attribute `Name`, and we have defined TYPE-IDENTIFIER in the TOKEN part of the language specification. All these definitions imply that type names in the specified language should be formed according to the lexical rules of TYPE-IDENTIFIER, and the string value of this TOKEN will be saved as type names in the schema.

2. The meta domain for the attribute `Inheritance` is a meta type INHERITANCE whose definition will be discussed later. It implies that the information of inheritance of a type definition is decided by the definition of INHERITANCE.

3. As we mentioned above, characteristics collectively represent the attributes, relationships and operations of a type. Similar to the `Names` attribute in OBJECT, the meta domain of the attribute `Characteristics` is a meta constructed type `list(CHARACTERISTICS)` which represents a list of `CHARACTERISTICS`.

   Here we explain some more about the meta type constructor `list`. Semantically it only includes the elements in the list. Syntactically, the elements may or may not appear together in the specified language. But for specification convenience, by default, the elements appear together and they are terminated or separated by a string. (We only need either terminator or separator for a list here. A separator is used to separate the elements in the list, while terminator is a terminator of the element in the list.) Here the default separator is ";" as specified in the above specification. (`sep` is an abbreviation of separator. We also have some other notation of meta type constructor list) Language specifiers can override all these defaults in the specification which we will explain later.

3.3.5 Examples of Modifications and Extensions

In Postgres, the type[3] create statement is in the following format,

```
"create"  <type name>
          "(" <characteristic> "," <characteristic> ... ")"
```

[3]In Postgres, the concept *class* is used instead of *type*, and a *class* has more meaning than a *type*, as we discussed above. For illustrative purpose here, we just treat a *class* as a *type*.

```
                [<inheritance>]
```

so the corresponding create meta operation and list separator for characteristics definition can be specified as follows:

```
TYPE::create("create", Name,
             "(", Characteristics,  ")", [Inheritance]);
TYPE::Characteristics.sep = "," ;
```

The difference between this create statement and the default one (at this level) are: first, in this create statement, the inheritance information is described at the end of the statement while in our default language, it is described immediately after the type name; second, the separator between each characteristic specification is "," here while in our default language, it is ";". Difference between these two type create statements may also show in other levels: at the level of characteristic specification and inheritance specification that will be described later.

3.4 INHERITANCE

3.4.1 General Discussion

As we explained before, INHERITANCE is an abstraction for inheritance specification between types. Inheritance is not a "first class" object in ordinary object models. It can not be defined independently and does not have any operation on it. Instead, it must be associated with type definitions.

INHERITANCE describes the relationship between a sub-type and its super-types. Multiple inheritance is now widely accepted, hence the sub/super type relationship is usually represented by a list of super-types specified in the sub-type.

Multiple inheritance may bring name conflicts in the case that more than one super-types have the same name of characteristics(i.e. attribute, relationship or operation). Right now, we, as many OODBMS's, suppose this kind of conflicts will be resolved statically by users, using renaming to specify from which the characteristics should be inherited.

Renaming can be used not only for the resolution of name conflict. Sometimes, because of the application vocabulary, the end user may prefer different names for the characteristics of the sub-type and super-type.

3.4.2 Basic Attributes

In our system, INHERITANCE has the following components. They are provided to the language specifiers to specify the corresponding features in their query languages.

- **Super-types:** which describes the immediate super types of this type. It is a list of type names;
- **Rename:** which describes new names(of attributes, relationships, and operations) should be used for the conflicting names. Usually it describes from which super type the renamed characteristic should inherit, and actually the new name can be the same as the inherited name as long as the semantics is clear. How to describe rename clauses is abstracted into the meta type RENAME, which is also described here.

3.4.3 Specification and Explanation

According to the above analysis, INHERITANCE component is defined in our meta language as following:

```
INHERITANCE {
  list(TYPE.Name)                  SuperTypes;
  list(RENAME)                     Renames;
  // default "constructor" for INHERITANCE.
  INHERITANCE(":", SuperTypes, ["rename", Renames]);
  // default separator or terminator for SuperTypes and Renames.
  SuperTypes.sep = "," ;
  Renames.sep = ";"
}
```

and RENAME is defined as following:

```
RENAME {
  TYPE.Name                        SuperTypeName,
  CHARACTERISTIC.Name              SuperTypeCharasName;
  CHARACTERISTIC.Name              NewCharasName;
  // default "constructor" for RENAME.
  RENAME([([(SuperTypeName, DOT)], SuperTypeCharasName)],
          "as", NewAttributeName);
}
```

There are several points we should explain for the above specification:

1. As we mentioned before, INHERITANCE is a non-first-class object, and the descriptions of inheritance in specified languages (i.e. constructing the inheritance components) are closely related to the type definition. For all these kinds of non-first-class objects, we also have the default built-in operations for them and the operation names are the components' names themselves (corresponding to constructors). For example, we have the above INHERITANCE and RENAME operation specifications.

2. When we specify the domain of `SuperTypes`, we used the *Meta Path Expression*: `TYPE.Name` in the meta constructed type `list(TYPE.Name)`, which means that in the list, every element should be a type name. This not only specifies that the domains of the elements are TOKENs (or more specific according to our default definition, TYPE-IDENTIFIER) since the domain of `TYPE.Name` is TOKEN(or TYPE-IDENTIFIER), but also put some kind of semantic constraints on the meta domain, i.e. these TOKENs must be type names, not just any string that conforms to the lexical rules of TOKEN(or TYPE-IDENTIFIER). According to this specification, we can generate some constraints checking semantic actions for this meta domain. The same idea for the domain definition of `SuperTypeAttrName` in RENAME definition.

3. As we mentioned above, CHARACTERISTIC collectively represents attributes, relationships and operations of a type. It is actually a meta super type of the meta types ATTRIBUTE, TRAVERSAL-PATH(representing relationship) and OPERATION. `Name` is an attribute of CHARACTERISTIC, that is why we can use `CHARACTERISTIC.Name` as a *Meta Path Expression* in defining the meta domain of `SuperTypeCharasName` and `NewCharasName`.

4. DOT in the RENAME constructor definition is another meta object in our system. It represents the connector in the path expression in the specified language. Usually, it is a dot("."). But some systems may want to use other form. The point of this definition here is to keep consistency. All the path expressions in the language should conform to the same format. DOT will be defined later in our system.

3.4.4 Examples of Modifications and Extensions

In our above examples for type definition, we did not explain what the inheritance looks like. Now, according to our default specification, we can see that the `<inheritance>` should be in the following format:

```
":" <type name list> ["rename" <rename clause list>]
```

and the `<type name list>` looks like:

```
<type name> ["," <type name> ...]
```

`<rename clause list>` looks like:

```
[<supertype name> "."] <old charas name> "as" <new charas name>
 [";" [<supertype name> "."] <old charas name> "as" <new charas name> ...]
```

where `<old charas name>` means the old characteristics name of one of the super type, and `<new charas name>` represents the new name that will be used in the sub-type.

In some systems, the inheritance is specified in some other ways, for example, in O2, we can specify the inheritance as follows:

```
INHERITANCE::INHERITANCE("inherits", SuperTypes, ["rename", Renames]);
```

which only differs in a very minor way, so the `<inheritance>` will look like as follows:

```
: "inherits" <type name list> [<rename clause list>]
```

And for SQL3, we can have the following specification:

```
INHERITANCE::INHERITANCE( "UNDER",
              { ( SuperTypes.ele,
                   [ "(" , "WITH", { Renames.ele
                       // attached actions to Renames.ele.
                       {Renames.ele.SuperTypeName = SuperTypes.ele},
                                    s";" }
                   ")" ]
                ), s","
              }     );
RENAME::RENAME(SuperTypeAttrName, "as", NewAttributeName);
```

This specification is more complicated than the above specifications. Basically in SQL3, the specification of inheritance looks like:

```
UNDER <type name> ["(" WITH <old charas name> AS <new charas name> ...")" ]
  [ "," <type name> ["(" WITH <old charas name> AS <new charas name> ")" ] ]
```

In our above specification, we use `SuperTypes.ele` to represent an element in the `SuperTypes` which is a meta list of type name. So `SuperTypes.ele` represents a type name that will be in the super-type name list of this type. The same idea applies to `Renames.ele`.

We also used a structure, *List Comma Expression* to represent the repeat structures in the specified language. For a *List Comma Expression*, we have two parts: the first one is the element part which can be any kind of *Comma Expression* in our system; and the second part specifies the separator/terminator for the list element, which in the form of `s"string"` or `e"string"`. So the string after `s` or `t` represents the separator or terminator respectively. In our above example, `Renames.ele` is the element part of the *List Comma Expression* (at this moment please ignore the part of attached actions), and `s";"` means that the separator for the element in the list is `";"`.

Another *List Comma Expression* in our above specification is more complex. The element part is an *Aggregational Comma Expression* which consists of a pair of `( )` and a sequence of *Comma Expressions.* Here the sequence of the *Comma Expressions* in this *Aggregational Comma Expression* are `SuperTypes.ele` and an *Optional Comma Expression* (which again is an *Aggregational Comma Expression*). The separator for the element in this *List Comma Expression* is `","` as specified above.

In our above specifications, we used a lot of meta type constructor `list`. The actual meaning of a meta list attribute is equivalent to a *List Comma Expression* in the sense that whenever the meta attribute (for example `Characteristic` in TYPE) appears in an meta operation (for example in `TYPE::create`), it can be expressed in a *List Comma Expression* consists of a pair of `{ }` and two parts in `{ }`: the first is the attribute name followed with `.ele` (for example:`Characteristic.ele`), the second part is either a `s` or at followed by a string of separator or terminator of the list.

The semantic action part for the *Comma Expression* `Renames.ele` is straight forward. Because the rename clause is immediately after the type (represented by the

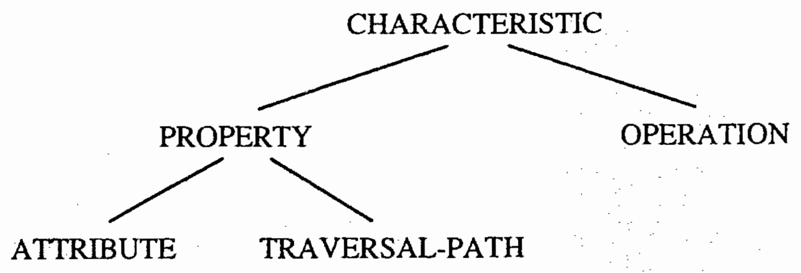

Figure 3.1: Characteristics subtree of the meta types

type name) of which the old name will be renamed, and in the rename clause the type name information is not specified anymore (The `RENAME::RENAME` is also redefined as in the above specification), we should record the type name in the `SuperTypeName` attribute of the rename structure. There are several other kinds of semantic action formats in our system which can be attached to a *Comma Expression* or a meta operation. We will explain these later with some examples.

3.5 CHARACTERISTIC and PROPERTY

As we mentioned above, attributes and relationships are collectively called properties, and together with operations, they are called characteristics of a type. To represent these concepts and the relationship among them, in our meta system, we have a meta type CHARACTERISTIC which has two sub meta types: PROPERTY and OPERATION. Under PROPERTY, we again have two sub meta types: ATTRIBUTE and TRAVERSAL-PATH (representing relationship). All these are illustrated by fig. 3.1

3.5.1 Basic Characteristics

The basic attributes of Characteristics are the the common attributes of attributes, traversal-paths and operations. They are *name* and *encapsulation tag*.

- **Name:** Name is the identifier of the characteristics. Characteristics names are not in the name space of the whole database. Instead, they are only unique within the type definition itself. We use the ordinary name scope semantics here. Syntactically, a characteristics name is also a string with a certain set lexical rules, which is usually described by the TOKEN of IDENTIFIER.

- **Encapsulation Tag:** Encapsulation and information hiding is another important concept in object-oriented paradigm. The basic idea is the separation of the interface from the implementation of the object. In ODMG-93, they separate type interface from its implementation. But they do not provide any concrete mechanism to do that, and to our knowledge there is no system available now that directly supports these concepts exactly. In our system, like many other systems, such as O2, Objectstore, SQL3, etc, we use the three level protection mechanism to do that. For each type, we have a *public* interface which is accessible by all the objects, a *protected* interface which is only accessible by the objects of sub-types, and a *private* characteristics which is only accessible by the objects of this type itself. Public interface definition is corresponding to the interface definition in ODMG-93, and private characteristics defines the implementation of the interface. Although ODMG-93 claim that they can provide more than one implementations for the same interface without abusing sub-types, we think there are some problems if a type has some characteristics corresponding to the *protected* interface. This is because the same object(type) may have different interfaces to different objects. Generally, we have objects of the type itself, objects of its sub-types, objects of friend types, and objects of others types. The interface may vary a lot between them. So the three level protected mechanism is widely accepted in most of the OODBMS's.

In our meta system, we use an encapsulation tag which is abstracted into the meta type ENCAPSULATION-TAG to represent this encapsulation mechanism. The tag can be any of the three levels, and the default one can be private, or public.

3.5.2 Specification and Explanation

According to the above discussion, we have the following definitions in our meta system:

```
CHARACTERISTIC {
    IDENTIFIER              Name;
    ENCAPSULATION-TAG       AccessTag;
}
ENUM ENCAPSULATION-TAG {
    KEYWORD                 Private;
    KEYWORD                 Protected;
    KEYWORD                 Public;
    // default format of the keywords
    Private("private");
    Protected("protected");
    Public("public");
    // special meta operation of defining the default(no tag).
    Default(Private);
}
PROPERTY : CHARACTERISTIC { }
```

There are several points we should explain about the above specification

1. For CHARACTERISTIC, we do not have any constructor, because its different sub-types may have their different constructors. But when we use CHARACTERISTIC, then it means that it can be any of its sub-types. For example, we used `list(CHARACTERISTIC)` in the meta type definition, which means that semantically and syntactically, the element in the list can be ATTRIBUTE, TRAVERSAL-PATH or OPERATION. When we use `CHARACTERISTIC.Name`, then the name of ATTRIBUTE, TRAVERSAL-PATH or OPERATION can be fit in.

   `PROPERTY : CHARACTERISTIC` means that PROPERTY is a sub meta type of CHARACTERISTIC. Here in PROPERTY, we do not have any more characteristics which should be defined. So its definition is empty.

2. As we discussed above, the access of any characteristic is controlled in the form of ENCAPSULATION-TAG, which is a meta enumerate type, as the LIFETIME we discussed above. Here we explain more about the meta enumerate type. Each attribute in the meta enumerate type is an element of the enumeration. In our above definition, the meta enumerate type ENCAPSULATION-TAG has three elements, and all of them are defined as KEYWORD. KEYWORD is again a sub type of TOKEN, which need to be specified by a special string used

as a keyword in the specified language. In our above specification, "private", "protected", and "public" are these special strings.

For the meta enumerate type, we also define a special meta operation `default`, which means that if none of the elements appears in the position, it is equivalent to the default element.

3.6 ATTRIBUTE

3.6.1 Basic Attributes

Attribute is not a "first class" in ordinary object models either. They must be defined for a type definition and exist with an object of this type. ATTRIBUTE is the abstracted structure for attribute definition. Since ATTRIBUTE is a sub-type of CHARACTERISTIC, it inherits all the characteristics of CHARACTERISTIC. In our following discussion, we will only discuss those special characteristics of ATTRIBUTE. Usually, an attribute definition includes the following aspects. So our ATTRIBUTE also includes the following components.

- **Name:** (the same as that of CHARACTERISTIC)
- **Domain:** Domain components of an attribute specifies the type and value range of the attributes. It is usually a type name either pre-defined in the system or predefined by users. It can also be a type construction.
- **Domain Constraints:** which describe some constraints on the attribute. There are two main kinds of domain constraint specifications. The simpler one is using some constraint tags, and different types may have different constraint tags. For

example, for the type of CHAR, it may have the constraints of "NOT NULL". Another one is using some constraint functions. For example, in ODMG-93, they can define *birthdate DATE CHECK(birthdate <> DATE'1992-01-01')*, which means that *birthdate* is of type *DATE* and with the constraint function *CHECK*. Right now, for simplicity, we only consider some constraint tags. Constraint functions can be analyzed in a similar way.

- **Default-value:** Default-value gives the attribute the initial value when the instance of this type is created. Default-value should be literals of the type of this attribute. Different types may have different literal construction formats, we will discuss these in the corresponding pre-defined types.[4]

3.6.2 Basic Operations

For attributes, we usually have two basic operations:

- **get-value():** which references the value. Usually we use the *rvalue* (right side value) of the attribute instead of this operation to retrieve the value, but we can also define this method.

- **set-value():** Similar to get-value(), deferences the value. Usually we use the *lvalue* (left side value) to represent this method.

In most systems, these two operations are represented by the "dot expression" uniformly. According to the context, the system can decide whether the "dot expression" is *rvalue* or *lvalue* of this attribute. "Dot expression" will be explained later in

[4]Right now, most of the systems assume one format of literal constructions for the user defined types. Our meta system currently also takes this assumption, although it is not very difficult to let users specify their special literal constructions for the types defined by them.

Chapter 4.

3.6.3 Specification and Explanation

According to the above discussion, we have the following definition in our meta system:

```
ATTRIBUTE : PROPERTY {
    TYPE                          Type;
    list(DOMAIN-CONSTRAINT)       ConstraintTags;
    LITERAL                       Default-value;
    ATTRIBUTE(Type, Name, ConstraintTags, Default-value );
    // default build-in operations for attribute
    LITERAL get-value(Name) ( "get-value", "(",  Name,  ")" );
    void set-value(Name, LITERAL) ( "set-value", "(", Name, LITERAL, ")" );
    ConstraintTags.sep=","
}
```

The following points should be explained for the above specification:

1. `ConstraintTags` is defined as a list of DOMAIN-CONSTRAINT, because for some types, there may be several constraints can be used together. DOMAIN-CONSTRAINT is a concept representing all possible domain constrain tags for different types. So it again depends on the language specification given by the language designers for the specified language. Our system will automatically collect all the domain constraint tags for the language specifiers. And at the same time, our system can check automatically whether the constraint tags are consistent or not with the type of the domain.

2. `Default-value` is defined as a LITERAL. The situation here is quite similar to the above situation for DOMAIN-CONSTRAINT. Since LITERAL is a concept representing all possible literal constructions in the specified language. The format of a literal depends on its type and also depends on the language specification. Our system will automatically collect all possible literal constructions in the specified language, and consistency (with the type) check can also be done automatically by our system.

The default built-in operation get-value() returns the value of this attribute in the specified language, and set-value() uses the value in the parameter to set the value of the attribute. The format of the function in the default language is specified in the the default specification for these two operations.

3.7 RELATIONSHIP

A relationship, like an attributes, is one of the characteristics of a type. It is defined between two types of objects, and can be used to traverse from the object of one type to the objects of the other, or vice visa, by the traversal path of the relationship. It maintains the referential integrity because if an object in the relationship is deleted, then it is automatically removed from the relationship so that there will not be any dangling object in this kind of reference.

A relationship does not have a name itself, but the traversal paths of the relationship have names.

3.7.1 Basic Attributes

Meta type TRAVERSAL-PATH in our system is the abstract structure for traversal-path definition in the specified query language, and it is consisted of the following components: (Again, we omit the characteristics that are inherited from CHARACTERISTIC)

- **Name:** (inherited from CHARACTERISTIC.)
- **Type:** The traversal path consists of the objects of the other type in the relationship. So the type of ***traversal path*** can be the other type in the relationship, or some kind of collection of the objects of the other type in the relationship.
- **Peer Type:** To pair two direction traversal paths of the relationship, we have to point out the inverse traversal path in the paired type. And actually, without the inverse path specification, it will only be an ordinary attribute, since the system has no way to maintain the reference integrity of the other side of the relationship. Peer type specifies the type in which the inverse traversal path is defined.
- **Inverse Path Name:** This specifies the traversal path name in the paired type that is the other part of the relationship.

Although there are a lot of different operations on relationship internally in most OODBMS, all the operations are not visible to end users. And traversal-paths are used just as ordinary attributes. So ***get-value***, ***set-value*** and ***"dot expression"*** can actually be specified for the PROPERTY so that both ATTRIBUTE and TRAVERSAL-PATH can inherit these operations.

3.7.2 Specification and Explanation

According to the above discussion, we have the following definition in our meta system:

```
TRAVERSAL-PATH : PROPERTY {
    TYPE                                    Type;
    TYPE.Name                               PeerType;
    TRAVERSAL-PATH.Name                     PeerName;
    // constructor of a traversal-path.
    TRAVERSAL-PATH(Type, Name, "inverse", PeerType, SCOPE-SIGN, PeerName );
}
```

Several points about the above specification is explained as follows:

1. Similar to the specification of ATTRIBUTE, the meta type of `Type` is TYPE. But the meta type of `PeerName` is only TYPE.Name. This is because it is only a name of a user (already) defined type and can not be any new type construction.

2. `PeerName` is defined as `TRAVERSAL-PATH.Name` because the paired name can not be anything else but another traversal path name. Some semantic actions can be generated to verify this.

3. `SCOPE-SIGN` is similar to `DOT` in our previous specification. It represents the symbol that used for the scope definition in the specified language. Usually it is "::" as in C++ or several other OO systems. `SCOPE-SIGN` will be specified later in Chapter 4.

3.8 OPERATION

3.8.1 Basic Attributes and Operations

In our system, we only consider the specification of the signature of an OPERATION. The implementation of the operation may be described in other kinds of programming languages, or in the query languages themselves. We will not go into details about that here.

Specification of an operation includes the following components:[5]

- **Return Type:** which specifies the type of the return value of the operation. It can be any type in the specified language.
- **Parameter List:** which specifies the parameters to the operation. Each parameter in the list is a parameter which is described by the meta type PARAMETER that is discussed as follows.

We also have some built-in operations on OPERATION, like invoke(), but they usually should not be visible to end users of the interactive query facility. Using the operation name with appropriate parameters is the simplest way to invoke the operation.

3.8.2 Specification and Explanation

```
OPERATION {
    TYPE                        ReturnType;
```

[5]Again, we omit the characteristics inherited from CHARACTERISTIC.

```
    list(PARAMETER)                Parameters;
    // default list separator:
    Parameters.sep = "," ;
    OPERATION(ReturnType, Name, "(", Parameters, ")");
    // default invocation operation on the OPERATION.
    invoke(Name, "(", {QUERY-EXP, s","}, ")" );
    // another format of invocation.
    invoke(Name, "(", {(Parameters.Name,  ":", QUERY-EXP), s","}, ")" );
}
PARAMETER {
    TYPE                           ParameterType;
    IDENTIFIER                     Name;
    // parameter passing mechanism;
    PARAMETER-PASSTYPE             PassType;
    PARAMETER(PassType,  ParameterType, [":", Name]);
}
EXCLUSIVE-ENUM PARAMETER-PASSTYPE {
    NO-TAG                         None;
    IN-OUT-TAG                     InOut;
    VAL-REF-TAG                    ValRef;
    // ENUM type: all, or one, here is one, and choose InOut;
    // if no choose function, then all.
    choose(InOut);
}
ENUM IN-OUT-TAG {
   KEYWORD                         In;
   KEYWORD                         Out;
   //default In, Out specification.
   In("in");
   Out("out");
}
VAL-REF-TAG ENUM {
   KEYWORD                         Value;
   KEYWORD                         Reference;
   //default Value, Reference specification.
   Value("by", "value");
   Out("by", "reference");
}
```

There are several points that should be explained:

1. In above specification, for the default invocation operation on OPERATION, we used a list of `QUERY-EXP` as the actual parameters to the operation. QUERY-EXP is a meta types defined for OQL which will be discussed in Chapter 4, representing all possible kinds of query expressions.

2. We used another kind of meta enumeration: `EXCLUSIVE-ENUM` to represent the exclusive choice of two different parameter passing mechanisms. Each element in this kind of meta enumeration is similar to the element in ordinary meta enumeration, but in the whole system, we can only choose one element in the specification, representing by the default operation `choose`.

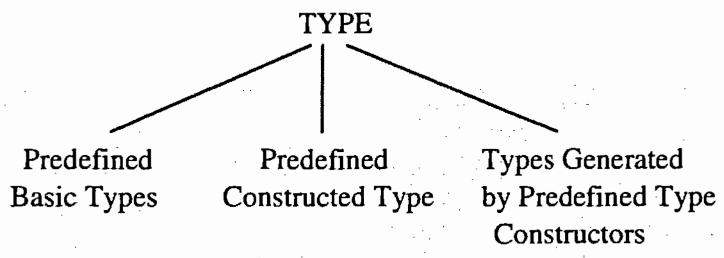
TYPE
Predefined Basic Types
Predefined Constructed Type
Types Generated by Predefined Type Constructors

Figure 3.2: Type System

3.9 TYPE, Type Generator and Type Hierarchy

As we stated before, all objects in an object-oriented system are classified into types, and all the types in the specified language form the type hierarchy of that language. In the above, we actually only analyzed the new type construction. But we can see from the above discussion that we need some basic types and type constructors for the attribute and relationship definitions in a new type construction. That is the reason why all the systems have to provide some pre-defined types, such as integer, character, etc, and some predefined type generators, such as array, structure, etc, by which end users can define more user-defined new types. All the predefined types and all the new types defined by end users consist of the whole type hierarchy of the application(see figure 3.2). In this section, we will study what kinds of predefined types and type generators a system may need, and also provide the abstract structures of them to language specifiers so that they can tailor or customize these components according to the needs of their specified languages.

3.9.1 Predefined Basic Types

Predefined types in a system usually include predefined basic types and predefined structured types. The predefined basic types are the lowest level in the type hierarchy.

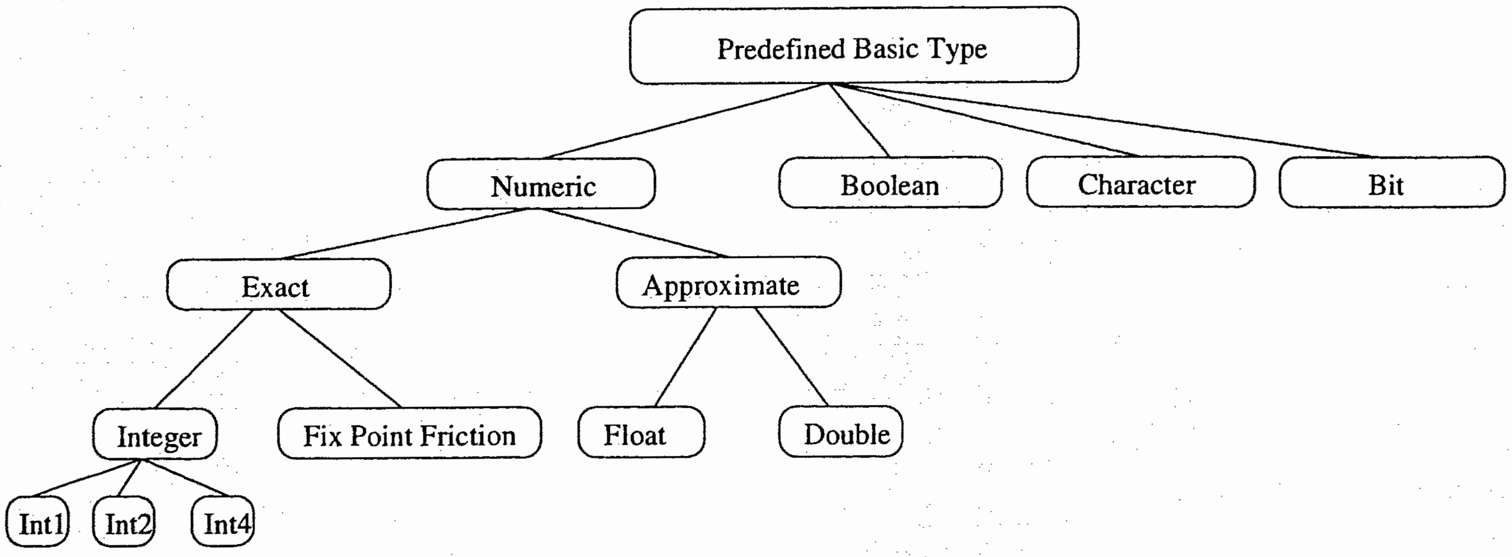

Figure 3.3: Basic Predefined Data Type

They are already implemented in the system for the end users and all the instances of these types are also pre-existed. End users will only use these instances and can not create any new instances of these types. All the operations on these types are also implemented in the system and end users can not redefine these operations. They can not define any new operations for these types, either.

Figure 3.3 illustrates the predefined basic types that we provide for language specifiers. All the inner nodes are *Abstract Data Type* (ADT) names and can not be instantiated directly while all the leaf nodes are the instantiable types supported by our system. But all these types can be customized by language specifiers.

Since there are so many different predefined basic types here, we can not analyze all the features of them in this thesis. Instead, we will use the *Numeric* subtree to illustrate our ideas about these predefined basic types and how language specifiers can customize them according to their needs.

Discussion on Numeric ADT

Numeric ADT has one common attribute: *Signed Tag*, which is used to specify the numeric range, i.e. whether the numeric instance is signed or not. All the arithmetic operations, for example, operators: $+, -, *, /$, etc are also defined here so that all its sub-types can inherit these operations. Since numeric instances are totally ordered, relational operators, such as $>, <, <=, >=, =$, etc are also defined for them.

Under *Numeric*, we have two sub-types, *Exact* and *Approximate*. Under *Exact*, we have *Fixed pointed Fractions*, which only a few systems distinguish them from float, and *Integers*, which in some systems may be divided into some sub-types according to the value range or the length of storage, such as *Int1*(1 byte), *Int2*(2 bytes) and *Int4*(4 bytes). Different systems may have different names of all these types, but we usually can customize the above type structures according to the specified languages.

Specification of Numeric ADT

Here are the specifications for the *Numeric* ADT and some of its sub-types.

```
Numeric : Predefined-Basic-Type {
  SIGN-TAG                SignTag;
  // operators defined on numeric ADT.
  Numeric operator+(op1:Numeric, op2:Numeric)
             (op1:QUERY-EXP, "+", op2:QUERY-EXP);
  Numeric operator-(op1:Numeric, op2:Numeric)
             (op1:QUERY-EXP, "-", op2:QUERY-EXP);
  Numeric operator*(op1:Numeric, op2:Numeric)
```

```
                (op1:QUERY-EXP, "*", op2:QUERY-EXP) );
   Numeric operator/(op1:Numeric, op2:Numeric)
                (op1:QUERY-EXP, "/", op2:QUERY-EXP);
   Boolean operator>(op1:Numeric, op2:Numeric)
                (op1:QUERY-EXP, ">", op2:QUERY-EXP);
   ...  ...
}
ENUM SIGN-TAG {
   KEYWORD                Signed;
   KEYWORD                Unsigned;
   Signed("signed");
   Unsigned("unsigned");
   default(Signed);
}
Exact : Numeric {
   TOKEN-INTEGER          Precision;
   TOKEN-INTEGER          Scale;
   Exact( SignTag, ( "NUMERIC" | "DECIMAL" | "DEC" ),
          [ "{", precision, ["," , scale], "}" ] );
   Exact( SignTag, ( "NUMERIC" | "DECIMAL" | "DEC" ))
          { Precision = 15; Scale = 6; }
   literal(TOKEN-INTEGER, [".", TOKEN-INTEGER]);
}
Integer : Exact {
   literal(TOKEN-INTEGER);
```

```
}
Int1 : Integer {
  Int1(SignTag, "tiny", "integer");
}
..., ...
```

Explanation and Examples of Numeric ADT Specification

Most of the specifications are similar to the specifications we described before, and we only explain the following new features here:

1. In the specification of the operators for `Numeric`, the format is more complex than other operation specification. The first part of the specification is exactly the same as the signature specification of C++ operator, which fully expresses the exact meaning of the operator. All the types in these part, although using the same name in our meta system, represents the types in the specified language mapped from the meta type in our meta system. For example, the return type `Numeric` represent the type in the specified language corresponding to the *Numeric* type in our meta system. Starting from the second parenthesis, specifications represent how these operators look like, similar to the other specifications we mentioned before. The only difference is that we give each parameter a name, and use the name to represent the relationship between the parameters in the original operator (or operation) signature and that in our meta operation. Usually, when the semantics (the original signature of the operation and the relationship between the parameters in the original operation and those in the meta operation) is clear, we always use simplified meta operation

specification format, as we used before.

2. There are two "constructors" for *Exact*. The first specifies the precision and scale for this exact fraction, while the second does not. But actually the second format of Exact type specification takes default values for the precision and scale. This is described by the semantic actions for the "constructor".

3. We defined a kind of special meta operation:`literal` to specify the literal format of this type. Actually this is an instance creation operation defined on these types. Because for all the predefined basic types, the instances of them are literals (i.e. instances without OIDs). That is the reason why we use `literal` as the meta operation name.

3.9.2 Predefined Type Generators

Only the predefined basic types are not enough for building a rich type system. Usually, an object-oriented system also provides several predefined type generators. Hence in our system, we also provide the corresponding abstract components for type generator specification. Figure 3.4 illustrates the basic hierarchy of our meta type generators. Usually, an enumeration generates a type which defines a list of names, any instance of this type will only have the value of these names, and have no other properties. Operations defined on the enumeration type are based on the equality of the names themselves and the total ordered relationship between the names in the list. Our meta type generator ENUMERATION embeds the basic semantics of general enumerations, and provides some ways to customize the concrete form of enumeration in the specified languages.

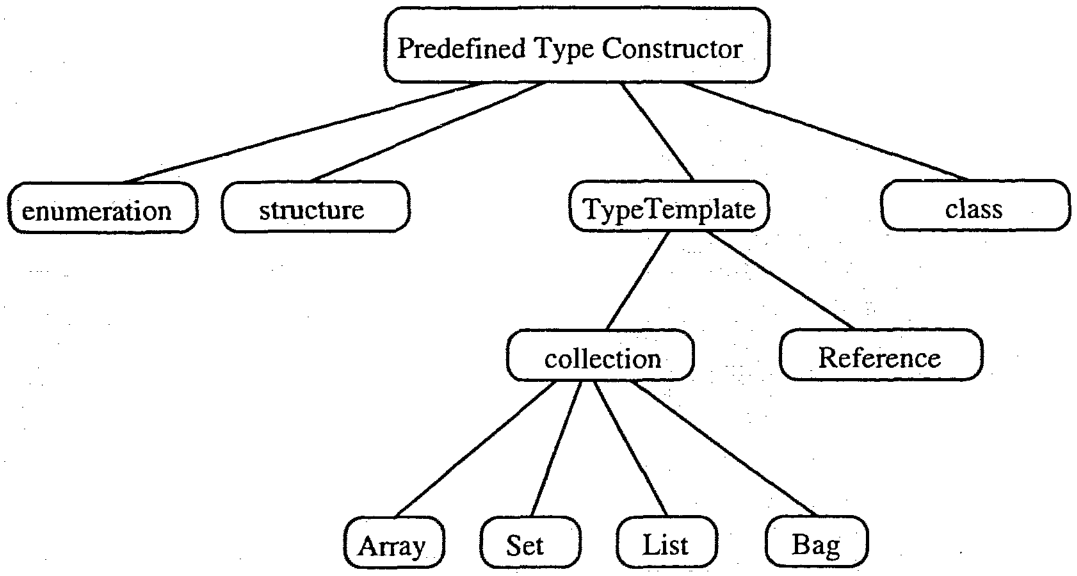

Figure 3.4: Predefined Type Generator

A structure generates a type which defines a fixed number of pairs of names and types. It is used to aggregate attributes, but is not like a new type definition in which other characteristics like relationships or operations are usually also defined. General operations defined on a structure are *get-value*, *set-value* of the member attributes, as for the attributes in an object type. Some systems also provide copy operation to copy one instance of the structure into another. In our system, a meta type generator STRUCTURE is provided for the specification of the specified type generator structure.

Template type generators, or parameterized types, are type generators which accept some types as parameters and then generate different types according to the different type parameters. Reference type generators and collection type generators are two kinds of commonly used template type generators and most of the object oriented systems provide them as predefined type generators. Some systems even allow user to define their own new template type generators. For simplicity, we do not consider this kind of functionality right now in our model.

Reference generators form types which define references to other objects of the specified types, usually the OIDs of that objects.

Collection generators are perhaps the most important type generators in an object-oriented data model. A collection object is used to group together other objects. All of which should be of the same type at some level. There are several attributes of a collection, like unordered or ordered, duplicated elements are allowed or not, etc. Because of the importance and complexity of collections, they will be discussed in a later section.

Specification and Explanation

Since the members of a structure are quite similar to the attributes in a type, and reference is rather simple, we will only describe the specification of ENUMERATION in our meta system:

```
ENUMERATION {
   TYPE-IDENTIFIER                          Name;
   list(IDENTIFIER)                         Elements;
   ENUMERATION("enum", Name, "{", Elements,  "}");
   BOOLEAN operator==(Elements.ele, Elements.ele)
             (Elements.ele, "==", Elements.ele) ;
}
```

There is nothing new to be explained in this specification .

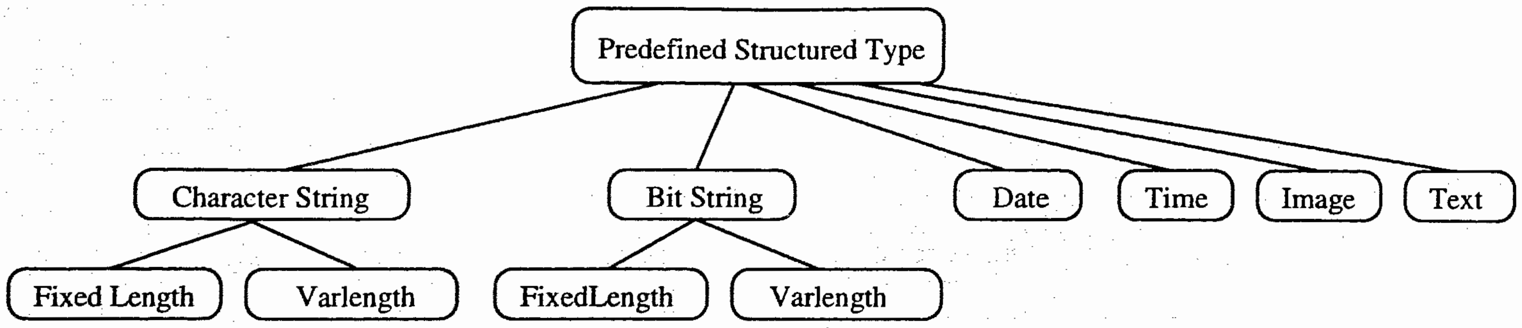

Figure 3.5: Predefined Structured Types

3.9.3 Predefined Structured Types

Most systems now provide not only the predefined basic types and predefined type constructors, but also some commonly used structured types (including some basic operations for these types) as predefined for the convenience of users. As illustrated by the Figure 3.5 in our system, we have a sub-tree of meta structured types that can be tailored or customized by language specifiers. But here for simplicity, we will not go into detail of the discussion and specifications about them.

3.10 COLLECTION

As we mentioned before, a collection object is used to group other objects together. Although in real situations, people seldom define collection types separately, collection type generators are heavily used in the domain definition for attributes or elements in the definitions of other types or structures. Defining named persistence collection object is also an important part in schema definition. This is because the names of persistence collection objects are often used as entry points to databases. Table definition based on an ADT in systems like SQL3 is actually a kind of named persistence collection object definition.

In this section, we will discuss both collection type definition and collection object creation. Collection object creation can be the same as other kinds of object creation in the specified language. But it can also be different, as we mentioned in the section on object discussion. Language specifiers can provide special collection object creation operations to override the default object creation operations, providing some initial values to the collection attributes, for example ordered tag, or other constraints.

3.10.1 Basic Attributes

We consider the following basic attributes of collections:

- **Name:** This is the name of the collection type. Usually a collection type is constructed without a name.

- **Element Type:** This is the type of the element in the collection. It is a parameter passed to the collection constructor when defining a collection type. Objects of sub-types of the element type can also be elements in the collection.

- **Cardinality:** This records how many elements in the collection. For the end users, this is a read-only attributes, and it is usually retrieved by a public function like count().

- **Ordered Tag:** This records whether the objects in the collection are ordered (usually according to the insertion or not, but it can also be ordered according to the values of some attributes)

- **Duplicated:** This records whether the collection allows duplicated elements or not.
- **Constraints:** This specifies some special constraints for the collection, for example, some fields must be unique, like keys, etc.

There are some other storage and performance related attributes that can be attached to the collections, for example, index, clustering pragma, etc. Again, for simplicity, we will not discuss these aspects in this thesis.

3.10.2 Basic Operations

Operations defined on a collection object are:

- **create:** Create a collection of the parameterized type. Some parameters which give some hints on performance, like clustering, may also be provided when the collection is created, but we will not consider this in our model right now.
- **delete:** delete the collection.
- **insert:** insert an object into the collection.
- **remove:** remove an object from the collection.
- **contain:** test whether or not a given object is in the collection.

3.10.3 Specification and Explanation

According to the above discussion, COLLECTION as the meta type abstracted for the collections in ordinary OODBMS's is defined as follows:

```
COLLECTION {
  TYPE-IDENTIFIER                     TypeName;
  TYPE                                ElementType;
  // instance properties:
  IDENTIFIER                          Name;
  Integer                             Cardinality;
  ORDER-TAG                           OrderTag;
  DUPLICATED-TAG                      DupTag;
  // a collection type creation
  create( "collection", "<", ElementType, ">");
  // collection instance creation.
  instance-create("create", "collection", "<", ElementType, ">",  Name);
  // operations on collection object, not collection type.
  void insert(ele:ElementType, col:COLLECTION<ElementType>)
           ("insert", ele:QUERY-EXP, "into", col:QUERY-EXP);
  void remove(ele:ElementType, col:COLLECTION<ElementType>)
           ("remove", ele:QUERY-EXP, "from", col:QUERY-EXP);
  Boolean operator in(ele:ElementType, col:COLLECTION<ElementType>)
           (ele:QUERY-EXP, "in", col:QUERY-EXP);

}
ENUM ORDER-TAG {
  KEYWORD              Ordered;
  KEYWORD              Unordered;
  default(Unordered);
  Ordered("ordered");
  Unordered("not", "ordered");
}
ENUM DUPLICATED-TAG {
  KEYWORD                     Duplicated;
  KEYWORD                     UnDuplicated;
  default(Duplicated);
  Ordered("dup");
  Unordered("not", "dup");
}
```

There are several points that should be explained for the above specification:

1. We define two different create operations. One is for collection type specification, which will be part of the type system; another is for collection instance creation, which actually creates a collection object.
2. In the signature of insert, remove, and the operator in, we used `COLLECTION <ElementType>` to represent the collection type generated according to the element type in the specified language. This is similar to the specification of the numeric operators.

3.10.4 Predefined Sub-types of Collection

There are some predefined sub-types of collection in most of the OODBMS's we studied. In our meta system, to represent these concepts, we also have the corresponding meta structures for these sub-types. They are Set, Bag, List and Array, as illustrated in the Fig. 3.5

A Set is an unordered collection that does not allow element duplication in the collection. Sets inherit and refine all the attributes and operations defined on collections and add some more operations usually defined as set operators: like union, intersection, difference, assignment, and the relational operators like subset, proper-subset, etc.

A Bag is a unordered collection that allows duplicated elements. So a bag should also maintain the number of occurrences of each element. All operations available to a set are available to a bag, but they are based on the numbers of the occurrences of the elements in the bag.

A List is an ordered collection which allows duplicated elements, and the order is maintained by the user when elements are inserted into the list. The default order is based on the insertion time. There is one more property defined on List, i.e the current-position as integer, which is similar to the cursor. There are also some other operations defined on List based on the position: such as insert-after, insert-before, remove-at, etc.

An Array is a one dimensional array with fixed or varying length. So we have some more properties about length defined on an array, such as the maximum length, etc.

All these sub-types of collection can be analyzed in a similar way as we did for collection. For simplicity, we omit these details in this thesis.

3.11 User Defined Components for ODL

From the above discussion we can see that the specification of a query language is a mapping from the concepts used in the specified object model and query language to the concepts in our predefined meta system. Sometimes, when a concept in the specified system is not easily mapped onto that in our meta system, the language specifiers can construct some new meta types for this concept. Constructing new meta types is quite similar to class definition in C++. For example, as we mentioned above, in some OODBMS's, the concept *class* which has both intent and extent is used instead of *type*, and until now we do not have a corresponding component defined for *class*. The following is our specification as a user defined meta type:

```
 CLASS {
     TYPE                                                   Intent;
     persistent COLLECTION<Intent>                          Extent;
     // constructors for the meta type CLASS:
     CLASS("class", Intent.Name, Intent.Inheritance, "{"
            Intent.Characteristics, "}" )
     {  Extent.Name = Intent.Name;  }
}
```

This means that a class consists of a type as the intent, and a persistent collection of its intent as the extent. The semantics of class creation is equivalent to the creation of this intent and extent; and the syntax is similar to type creation with the extent collection name the same as the intent type name.

Chapter 4

OQL and META-OQL

While the Object Definition Language provides the facilities of the database schema definition, the Object Query Language deals with the general object operations: such as object create, delete, update, and retrieve. Most of the operations on objects have actually been defined in the corresponding type definitions of the objects, and OQL basically provides some kind of command, function or operator format for these operations. Some statements may involve arbitrary types of objects, such as select-from-where statement. They have not been defined within any type of objects and should be defined specially in OQL.

In this chapter, we will discuss our common components abstracted from the Object Query Languages of ordinary object-oriented database systems. Again, we will not describe a complete abstract query language, but rather only include the basic aspects of object manipulation itself.

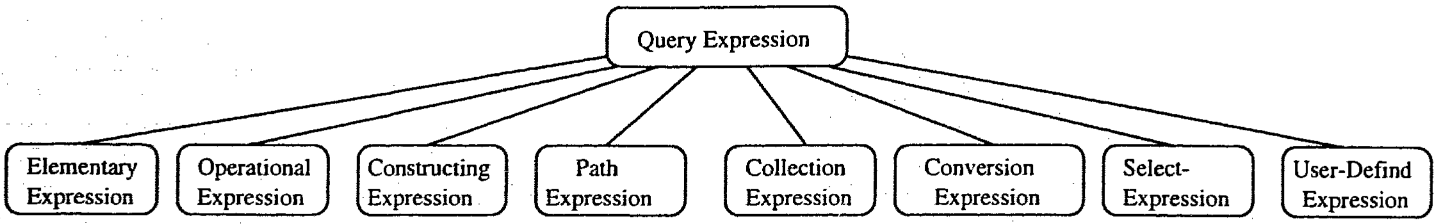

Figure 4.1: Query Expression Classification

4.1 General Discussion

Generally, object manipulation includes object creation, retrieval, update (such as changing the state of an object, inserting an object into a collection, etc). According to our studies of different query languages, we abstracted the meta type *Query Expression* to represent the basic unit of OQL. Furthermore, we classified *Query Expression* into the following sub-types which represent different kinds of query expressions: (see Fig. 4.1)

- **Elementary Expressions:** These kinds of expressions mainly include literals of the predefined basic types and object names.
- **Operational Expressions:** These kinds of expressions mainly include arithmetic, relational and logic operational expressions.
- **Constructing Expressions:** These kinds of expressions mainly include different kinds of object creation. For example, creating objects of user defined types, instances of some structures, sets, and lists, etc.
- **Path Expressions:** These kinds of expressions include expressions for accessing attributes, relationships, and operations of an object.

- **Collection Expressions:** These kinds of expressions mainly include expressions based on different types of collection objects. For example, how to manipulate sets, lists, etc.

- **Conversion Expressions:** These kinds of expressions include expressions of conversions between different types. This is one of the important parts of the type system for the OODBMS's.

- **Select Expressions:** These kinds of expressions are based on the select-from-where expressions in traditional SQL. It provides a mechanism to manipulate objects of more than one type (which are not directly related in the schema definition) in one statement.

- **User-defined Expressions:** These are the mechanisms for language specifiers to construct some special kinds of expressions for their CQLs. For example, the MOVE statement of DSQL.

From the above, we can see that most of the query expressions only deal with object creation and retrieval, none of them deals with update and delete. This is because we can build these statements or expressions based on the above query expressions. Hence in our later discussion, we will only concentrate on create and retrieval statements or expressions of OQL. A similar approach can be used for update and delete expressions, and we will not discuss these further.

In the following sections, we will discuss our common components for all of these sub-types of the *Query Expression*. A *Query Expression* can be formed recursively, as we will see later. For convenience, we will use `QUERY-EXP` to represent the meta type in our meta language (As we have already mentioned in Chapter 3). Each sub-meta-type of *Query Expression* also has a symbol in our meta language, which will be discussed in the following sections. But before the detailed discussion of these *Query Expressions*, we should have a general discussion about query languages and query expressions.

In traditional SQL for RDBMS, a data retrieval statement is usually only in the format of the select-from-where statement. In some OODBMS's, although they also include the select-from-where statement, for some simple situations they also allow some more concise formats of the expressions. For example, suppose we have an entry name, say Student, as the name of a persistence collection. Then the entry name Student itself forms a query expression, and the result of this query expression is the collection of instances of Student. Actually, we can let any of the above query expressions be a single object retrieval statement.

Usually, each query expression has a result, which is also the output of the query. Query results can be of all different types, such as instances of user defined types, literals of predefined basic types, constructed types, etc, depending on the query expression itself. Most of them are obvious, and we will explain this later when we discuss each sub-type of the *Query Expression.* But for select-from-where statement, the situation is more complex. In traditional SQL, the result of a select-from-where statement is also an ordinary relation. This property is usually referred as the "Closure Property". In some of the query languages of OODBMS, the result of a select-from-where statement can also be of different types depending on the result part of the statement itself. Here, it seems that the "Closure Property" is not retained. But actually, this is because in OQL there are more types which can be used in query expressions. The results of query expressions (including select-from-where) can be used in other query expressions as ordinary query expressions, as long as the types of the expressions are compatible with other expressions. We will discuss more about that in the section about select-from-where expression.

4.2 Basic Expressions

The basic query expressions include literals of predefined basic types, such as integers, strings, etc, and the object names, which are similar to variable names in ordinary programming languages. Whatever the format of the object names and the literals of the predefined basic types have already been defined as in the corresponding meta type definitions by the language specifiers (or if they use the default formats of our default query language), the concept of basic expression will include them. That is to say that a basic expression BASIC-EXP, includes:

```
OBJECT.Name //defined in meta type OBJECT.
<Literal-Exact> //defined in meta type Exact.
<Literal-Integer> //defined in meta type Integer.
..., ...
```

By default, a BASIC-EXP can form a statement in OQL. That is to say, whatever kind of formats of the object name or the literals of the predefined basic types, they will be one kind of query statement.

4.3 Operational Expressions

For the predefined basic types, we assume that we have already defined all the basic operations in the corresponding meta types. For example, for the `Numeric` ADT, we defined all the arithmetic and relational operators, and how to form the operational expressions, such as `QUERY-EXP + QUERY-EXP`, `QUERY-EXP - QUERY-EXP`, etc.

Operational expressions can be formed recursively, as we can see from the above operator definitions. The operands for these operators are `QUERY-EXP`s, which in turn can be any of the sub types of query expressions, including literals, operational expressions, path expressions, etc, as long as the types of the operands are compatible. So whatever kind of operators and expression forms are specified for the predefined basic types, they will be part of the operational expressions of the specified query language. Hence `OPER-EXP`, which represents *Operational Expression* in our meta system includes:

```
<Query-Exp> == <Query-Exp> // defined in meta type OBJECT.
<Query-Exp> + <Query-Exp> // defined in meta type Numeric
<Query-Exp> - <Query-Exp> // defined in meta type Numeric
...,  ...
```

All the operators for the predefined types have the default priorities as in ordinary programming languages, and parenthesis can be used to change the order of the calculations. Right now, we do not provide mechanisms for customization of operator priorities in our system, but it can be done (using priority orders) in a similar way as we did for other concepts.

4.4 Object Constructing Expressions

All the data and objects in the database should be created in some kind of format. In ordinary object-oriented programming languages, such as C++, objects and data are usually created by some data definition statements. In query languages, these are done through some types of "create" statements. No matter what kind of format it may be, the underlying semantics are quite similar. Usually, for object creation, the system will invoke the constructors, either user defined or system default, of the type for this object, passing some corresponding parameters.

There are two kinds of parameter passing styles for operations. In ordinary programming languages, the parameters are passed according to position. That is to say that the meaning of the parameters are decided by their positions in the operation parameter list. The users of the operations should understand and remember the position conventions specified by the implementation of these operations. While in some query languages, for object creation operations, the parameters are passed by the attribute name and value pairs, which is much easier for ordinary users to use. In our system, we provide both of these styles for object creation. For example, if we have defined type Student with some attributes such as name, student-no, age, etc. We can use

```
Student(name: "Mike", student-no:"94000-0000", age:20);
```

to create an instance of Student. If in the definition of Student, the user has defined a constructor in the format:

```
Student(char* st-no, int age, char* name) ;
```

then the statement

```
Student("94000-0000", 20, "Mike");
```

will also create the same instance of Student.

However, the above two kinds of object creation expressions should not be mixed in one operation. For each operation, only one style can be used. From the point of internal processing, these two kinds of parameter passing styles can be distinguished easily according to their "signatures" because we can treat the first kind of object creation as a special kind of operation with a special signature.

The second kind of object creation is just an ordinary operation invocation. So it must be based on the user defined constructors for this type. And the actual format of the operation invocation must be consistent with the operation invocation convention in the specified query language. In our system, the first style is the default object creation format for all the user defined types. The actual format of the invocation can be specified by the language specifiers in the meta type TYPE definition by the instance-creation operation as follows:

```
// default instance creation operation for all the user defined types.
TYPE::instance-create(Name, "(", {
          (Attributes.ele.Name, ":", QUERY-EXP),
                s"," }, ")");
```

For predefined types, we have discussed instance creation for the predefined basic types in an elementary expression (the literal definition is of this kind of operation). For predefined structured types, the creation of a structure instance is similar to that of object creation. Since there are no user-defined constructors for the instance of a structure, and the instance of a structure type is literal, we can only provide a general literal creation operation for all user defined structure types.

```
// default literal constructor for all the structure types.
STRUCTURE::Literal(Name, "(", {
    (Members.ele.Name, ":", QUERY-EXP), s","
                                        }, ")");
```

Sometimes, we may need to dynamically create a temporary structure type (with or without a type name) and then create instances of this type. For example,

```
struct(name: "Mike",  age:20);
```

may be used to create a no name type of structure, and also create a literal instance of this type. And

```
struct person(name: "Mike",  age:20);
```

will create a structure of type `person`, and at the same time create an instance for this type.

The operation of dynamically creating a temporary structure type is very useful for storing the result returned by a complex query, especially when used in the select-from-where statement. We call this operation temporary structure type and instance creation. The actual format of this can be specified by the language specifier under the meta function name `temp-struct`. And the default format is:

```
// default literal constructor for all the structure types.
STRUCTURE::temp-struct("struct", [Name,] "(", {
   (Members.ele.Name, ":", QUERY-EXP), s"," }, ")");
```

For other predefined structured types, such as list, set, bag etc, the constructing expressions are defined in their instance-creation meta operation in the corresponding meta types, and we will not go into details of them here.

4.5 Path Expressions

Path expressions are used to access the characteristics of a type. Usually there are two types of path expressions: one is so called a *dot expression*, and the format is `OBJECT.Name DOT CHARACTERISTIC.Name` where `DOT` is usually specified as `DOT(".")` in the meta system. Another one is so called a *scope expression*, and the format is `TYPE.Name SCOPE-SIGN CHARACTERISTIC.Name` where `SCOPE-SIGN` is usually specified as `SCOPE-SIGN("::")`. Language specifiers can specify their own format of `DOT` or `SCOPE-SIGN` themselves if they like.

4.6 Collection Related Expressions

Collection Related Expression basically have been defined in the meta type COLLECTION and its subtypes, such as SET, LIST, etc. For example, we have defined the operator **in** to test the *contain* relationship between an element and a collection instance in the meta type definition of COLLECTION. Here we will not discuss these operations further.

4.7 Type Conversion Expressions

Although, according to the principles of the object-oriented approach, different types should not be mixed in a calculation, sometimes users may still want to convert an instance of one type to an instance of another type in a reasonable way. Usually type conversions are only within the predefined types (including the predefined basic types, predefined structured types and types constructed through the predefined type constructors) and the conversion are also predefined in some reasonable way. This is because it may not make sense if we permit conversions between arbitrary types.

Type conversion operations are defined on two different types, and usually they are treated as some kind of top level meta operation, instead of an operation of one type. Here we will not enumerate all possible type conversion operations in an ordinary query language, but rather use one example: the operation of converting a bag into a set, to illustrate the basic ideas of this kind of expression:

```
// default bag to set conversion operation.
Bag Bag2Set(Set) ("bagtoset", "(", QUERY-EXP, ")");
```

The meta definition of this meta operation is quite similar to the operations we discussed in the `Numeric` ADT and we will not give any more explanation here.

Users can also construct new type conversion operations between any two different types. Since it is the same way as constructing other new operations, so we will not discuss this further here.

4.8 Select Expressions

4.8.1 General Discussion

In traditional SQL, the select-from-where statement is the main data retrieval statement. Because of the popularity of SQL, most query languages of OODBMS's also have similar structures, and usually use similar syntax appearance. Basically, this structure reflects the following three aspects of a data/object retrieval statement:

- **Result Clause:** This clause describes what kind of data/object must be retrieved by the query statement. In traditional SQL, this part consists of a list of expression, and each expression is only based on the data whose level is no more than column level (no structures or relations can be used here). But in most query languages of OODBMS's, all different expressions or sub-query expressions can be used here as long as the computation is permitted by the type system.
- **Range Clause:** This clause describes against which part of the database the query is done. In traditional SQL, this part binds the tuple variables (if there is any) to the relations, specifying against which relations the query is issued. In object-oriented query languages, this clause also binds different variables to different domains which usually are collections. Again, sub-query expressions can be used here as long as the type system permits.
- **Condition Clause:** This clause describes what kind of data/objects are qualified for the result, and it is just like a filter. Usually it is a kind of logic expression in which different predicates may appear. In a broad sense, it is a

kind of constraint specification of the variables appearing in the result and range clauses.

In some cases, the range clause can be omitted if from the result clause we can know against which part of the data we are issuing the queries. A condition clause can also be omitted when there is no condition specified. For example, if we have a persistent collection named Student, `select Student` itself is very clear about the range and condition.

4.8.2 Basic Semantics

The basic semantics for the select-from-where statement is that, for each range variable (including all the implicit range variables in the statement), we will generate a foreach loop (nested if some loops have been already generated) to scan all the instances in the collection of this range. In the inner loop, we check the conditions of the condition clause, and filter out a subset (or sub-bag [1]) of the cartesian product of all the range variables that satisfy all the conditions. We then evaluate the query expressions in the result part for all the filtered instances of the cartesian product.

```
foreach v1 in col1
   foreach v2 in col2
      ...
         if ( cond(v1, v2, ...) ) {
                r(v1, v2,...)
```

[1]If any collection is a bag, then the result may also be a bag. The cartesian product of bags is defined similar to the cartesian product as set. For example, col1 = { 1, 2, 3 }, and col2 = { a, b, a }, then the cartesian product of col1 and col2 is { (1, a), (2, a), (3, a), (1, b), (2, b), (3, b), (1, a), (2, a), (3, a) }

```
        }
```

Here, the expressions `cond(v1, v2, ...)` and `r(v1, v2,...)` represent the condition and the result evaluated with the range variables replaced by the current instances in the collections.

The above code schema only represents the basic semantics of the select-from-where expression. Usually some optimizations must be done to make the code more efficient. For example, if a sub condition expression in the condition clause is only restricted to one collection and is easily be tested, then we may first evaluate this condition, and get another smaller collection `COL1` to replace the original `col1`. Some optimization strategies for relational queries are still useful for object-oriented query expression evaluations, but in many cases the optimization algorithms for object-oriented query expressions are more complex than those for the simple traditional queries. In this thesis, we will not discuss this aspect of query language processing.

4.8.3 Specification

According to the above discussion, we have the following specification for the select-from-where expression:

```
SELECT-EXP {
    list(QUERY-EXP)              Result;
    list(RANGE-EXP)              Range;
    list(QUERY-EXP)              Condition;
    // default constructor for the SELECT-EXP;
    SELECT-EXP("select", Result, ["from", Range], ["where", Condition]);
}
RANGE-EXP {
   IDENTIFIER               Range-var;
   QUERY-EXP                Domain;
   RANGE-EXP(Domain, [":", Range-var]);
   // other format: ODMG-93
   RANGE-EXP(Range-var, "in", Domain );
}
```

The specification here is similar to those specifications we presented above and no more explanation is needed.

4.9 User-defined Expressions

As we mentioned above, language specifiers can construct some new statements to represent the new components which are not easily expressed in the basic meta components we provided. A new component is usually in the format of a new statement, and in the new statement specification, language specifiers describe how the new statement can be mapped on some basic meta components. Here we use an example taken from DSQL and specify the "move" statement as follows:

```
NEW-STATEMENT::MOVE("move", TYPE.Name:t1, ":", IDENTIFIER:v1
               "to", TYPE.Name:t2, ":", IDENTIFIER:v2,
               SELECT-EXP.Conditions:c1)
{
  foreach ( (v1, v2) in
            (select v1, v2 from t1:v1, t2:v2 where c1) )
        move(v1, v2);
  }
```

We assume that the language designers provided their own underlying "move" function here. This specification actually mapped the new statement "move" to the meta structures we provided in our meta system. `foreach` is a meta structure representing a loop over a collection object, and the `select` statement represents the meta select-from-where structure in our system that will return a collection. Using the named parameters, the relationship between the parameters in the new components and the parameters in the meta structures can be revealed clearly. This idea is similar to those we used in Chapter 3.

Chapter 5

System Implementation

After the above discussion about our design, in this chapter we will give a more detailed discussion of how our system works and how it can be implemented. Since a complete language definition and practical implementation of our system would have too many technical details, we will use some small examples to show how a language can be specified in our system, and how the specification of the language can be used to generate a query processor for this language. The implementation described here is based on our test implementation of our system, which is mainly based on YACC/LEX, C, C++ and one OODBMS: Objectstore.

5.1 Examples

The examples we will use are taken from [5], which are basically two typical class definition statements in O2.

```
class City
```

```
    type tuple(name: string,
               map: Bitmap,
               hotels: set(Hotel))
    method how_many_vacancies(star: integer): integer,
           build_new_hotel(h: Hotel)
end;

class Tourist_City inherit City
    rename build_new_hotel as new_equipment
           /* rename to a more appropriate one */
    type tuple(hotels: set(Hotel_Restaurant), /* attribute overriding */
                what_to_see: set(monument)) /* new attribute */
    method new_equipment(e: Hotel_Restaurant)
             /* method overriding (build_new_hotel in fact) */
end;
```

The meaning of the examples is quite obvious. The first statement defines a class named `City` with 3 attributes and 2 methods. The second statement defines a class named `Tourist_City` which is a sub-class of `City`. One more attribute is defined for this sub-class. It also redefines one attribute `hotels` and one method **`build_new_hotel`**. In the following, we will describe the specification for this part of O2 based on our system. The semantics we assume here may not be strictly consistent with the original O2 specification because of our illustrative purpose. We will point out these difference when it is necessary later.

In the following specification, the numbers before each meta statement are used for reference to our explanation. They are not part of the specification. Neither is the comments.

```
    (1)  TOKEN  TYPE-IDENTIFIER  [A-Z]{letter-or-digit}*
%%
    (2)  CLASS::create("class", Type.Name, [Type.Inheritance],
              "type", "tuple", "(", Type.Attributes, ")",
               [Type.Operations], "end");
    (3)  TYPE::Name(TYPE-IDENTIFIER);
    (4)  CLASS.Type.Attributes.sep = "," ;
    (5)   INHERITANCE::INHERITANCE("inherits",
                SuperTypes, ["rename", Renames]);
    (6)  OPERATION::OPERATION("method",  Name,
                "(", Parameters, ")", ":" ReturnType);
         PARAMETER::PARAMETER([VarName, ":"], ParameterType);
    (7)  ATTRIBUTE::ATTRIBUTE(Name, ":", Domain);
    (8) /* predefined types and type constructors */
          (a)    VARCHAR::VARCHAR("string");
          (b)    VARBITSTRING::VARBITSTRING("Bitmap");
          (c)    SET::SET("set", "(", TYPE, ")");
%%
```

Meta statement (1) is a token definition example in our system. It defines a token `TYPE-IDENTIFIER` which starts with an upper case letter and follows with letters or digits. This token definition is used in (3) which specifies that a type name should be in the format of this token. In O2 this is not the case, here we use this as an example of defining tokens and how to use the defined tokens.

"%%" after the token definition is a kind of separator between the first part (token definition) and the second part of a language specification. It is a kind of syntactic sugar in our meta language and has a simiiar function as that in YACC programs. The second "%%" is similar.

Meta statement (2) is the overriding of the class (not type) creation meta operation (the meta type CLASS and the meta operations defined on CLASS are defined in Section 3.11) which actually specifies how a class definition should look like in O2.[1]

INHERITANCE specification in meta statement (5) is almost the same as the default one except that in O2 they use "inherits" instead of ":" to lead the inheritance definition. The rename clause is also the same as the default one (please see section 3.4.3.)

Meta statements (4) and (7) specify how attributes (see Section 3.6.3) of a class should be defined in O2. The separator between the attribute definitions is "," instead of the default ";". And for the definition of each attribute, the format is defined as (7) instead of being the default one.

Meta statement (6) specifies how operations of a class can be specified in O2. They are overriding the predefined constructors of the meta type OPERATION and PARAMETER (see Section 3.8.2).

[1]For the simplicity of our example, we use this kind of specification here. Again, it may not be strictly consistent with the actual semantics of the O2 language. A better specification can keep the concept of tuple and map it onto the structure in our system, but we will not go into details of this here.

Meta statements in (8) specify how the predefined types and type constructors in O2 should be mapped onto the predefined types and type constructors in our abstract Object Model. Each of them corresponds to the overriding of the constructor of the corresponding meta type. (The specification is similar to other meta type constructors.)

5.2 How Our System Works

From the above example we can see that, since we have provided a general abstract Object Model and Default query language, ordinary query languages for object oriented database systems can be specified based on our system in a simple way. As we mentioned above, since query language specifiers can reuse a lot of common components provided by our system, usually what they have to do is to provide some special feature descriptions for their query languages. This kind of specification is usually much shorter and clearer than an ordinary language specification.

5.2.1 Language Specification

Basically, a language specification consists of the following three kinds of specifications (see figure 5.1).

1. **Token Definition:** provides the special lexical elements (token definition using regular expressions) for the specified languages. In the above example, meta statement (1) belongs to this category. In our system, we have already defined some default tokens (with the default lexical rules) for the language specifiers,

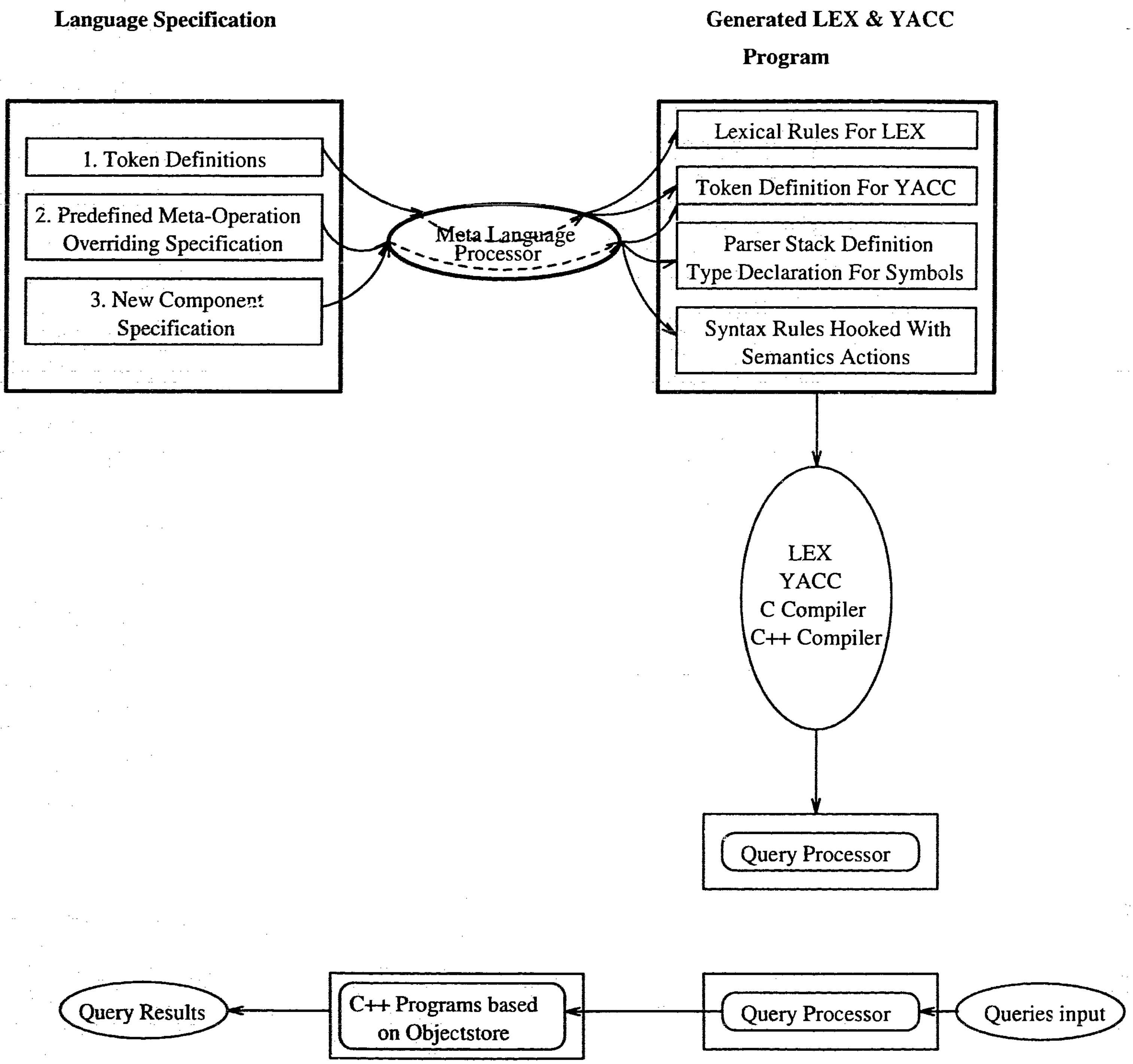
Language Specification
1. Token Definitions
2. Predefined Meta-Operation Overriding Specification
3. New Component Specification
Meta Language Processor
Generated LEX & YACC Program
Lexical Rules For LEX
Token Definition For YACC
Parser Stack Definition Type Declaration For Symbols
Syntax Rules Hooked With Semantics Actions
LEX
YACC
C Compiler
C++ Compiler
Query Processor
Query Results
C++ Programs based on Objectstore
Query Processor
Queries input

Figure 5.1: How Our System Works

such as TOKEN-INTEGER, which represents the digit string, and TOKEN-IDENTIFIER, which represents how an identifier should be composed. Language specifiers can directly use these default tokens, but they can also provide their own token definitions. Since right now our system implementation and part of the presentation of our system design are based on the YACC and LEX programs, in some way our system is influenced by YACC and LEX. We use the same token definition conventions as YACC ([21]) and LEX ([22]). LEX programs can also be used with our system because of this reason [2]. As in YACC, token definition is the first part of our specification and is separated from the next part of the specification by %%.

2. **Overriding Predefined Meta Operations:** According to our design, most of the basic concepts of Object Model and query languages are defined in meta types/objects. If language specifiers want to customize or tailor the semantics/syntax of the query languages, they can override the default meta-operations (usually the constructors) defined in the corresponding meta types/objects. Actually, providing these kinds of meta operations is one of the main kinds of language specification. In the above example, most of the meta statements belong to this category. Statement (3) and (4) provide overriding of an attribute domain and default separator of a meta list. They are also used in generating YACC programs for the specification.

3. **Constructing New Components:** Language specifiers can also define their own meta types and meta operations (either for some meta types or not) using the predefined meta types for some special components in their query languages

[2]So we can declare the tokens without providing the regular rules in this specification and provide a LEX program to our system.

in a similar way of the definition to the predefined meta types (see Section 3.11 and Section 4.9). After being defined, these meta types and objects can be used in the same manner as the predefined ones. New component constructions are another main kind of language specification. They can be mixed with the specification of overriding the predefined meta-operations.

5.2.2 Processing the Language Specification

From the above three kinds of specification provided by the language specifiers, we can then generate YACC and LEX programs. With YACC, LEX, C, C++ compilers etc, we can further generate a query processor for the specified query language automatically. The whole process is illustrated in figure 5.1.

The token definition part will be transformed into a LEX program segment and then merged into our predefined LEX program (in which some keywords and other default token definitions are dealt with). It will also be used to generate part of token definitions that will be merged into the token definition part of our YACC program. These two kinds of transformation are quite straight forward because we use the same conventions as LEX and YACC. Here we only use the above example to show the basic idea of this transformation without going into more technical details.

In our predefined LEX programs, we have the following LEX definition:

```
...., ...
letter                      [a-zA-Z_]
digit                       [0-9]
letter_or_digit             [a-zA-Z_0-9]
```

```
    white_space                    [ \t\n]
    ..., ...
```

Language specifiers can use these definitions to define their tokens in regular expressions. In the above example, we have

```
TOKEN  TYPE-IDENTIFIER  [A-Z]{letter-or-digit}*
```

We can generate the following LEX program segment from the above definition:[3]

```
[A-Z]{letter-or-digit}*
  {
    yylval.str = malloc(strlen(yytext)+1);
    strcpy(yylval.str, yytext);
    return token(TYPE-IDENTIFIER);
  }
```

which will be merged (or replace the corresponding default token definition) into the rule part of our predefined LEX program.

The above definition also generates the following YACC program segment

```
%token <str>                  TYPE-IDENTIFIER
```

which will be added to the token definition of the YACC program generated by our system.

[3]In our test implementation, the user defined tokens are of the type of string, so we only save the string value in the parser stack of YACC. See the following YACC program segment generated by the above definition.

The main part of the whole process is to generate the two main parts of the YACC program from the operation specifications and new component constructions of the specification: (1) syntax rules and semantics actions hooked to the rules; (2) parser stack definition and type definitions for symbols appearing in the syntax rules. We will describe the algorithm for these parts of our system in the next section.

Right now, the query processors generated by our testing system work in a different way from traditional query processors. Queries input by the users will first be processed and transformed into C++ programs based on Objectstore OODBMS; then the programs will be compiled into a dynamic link library. Dynamically linked and run with our predefined main program, query results for the queries can be produced and returned to the users.

5.3 Implementation of Our System

In this section, we will describe our basic algorithms of how to generate YACC programs from the specification of overriding the predefined meta-operations and constructing new components. This process itself is again based on the YACC and LEX tools. That is to say, we will use YACC and LEX programs to generate YACC and LEX programs.

5.3.1 BNF for the Meta Operations

In our system, the basic unit of the specification is in the form of so-called *"meta-operations"*. The form of this meta-operation specification is similar to C++ function prototype definition. The difference is that the parameters to the meta-operation are in the form of *"Comma Expressions"* which are recursively defined as following:

- A string enclosed by double quotes is a *Comma Expression*, which represents the direct TOKEN in the target language (i.e. this component will match exactly the string in the quotes in the target language). This kind of *Comma Expression* is called a *String Comma Expression.*

- A *Meta Path Expression* is a *Comma Expression*, which represents the concept represented by the *Meta Path Expression* in our meta language. It usually represents another component in our meta language. This kind of *Comma Expression* is called a *Path Comma Expression*

- A *Comma Expression* enclosed by '[' and ']' is a *Comma Expression*, which represents that this parameter may or may not appear in the target language. This kind of *Comma Expression* is called an *Optional Comma Expression*

- A *Comma Expression* and a *String Comma Expression* separated by a comma and enclosed by '{' and '}', is also a *Comma Expression* which represents a list of the *Comma Expression* separated by the *String Comma Expression.* This kind of *Comma Expression* is called a *List Comma Expression*

- A list *Comma Expression* separated by vertical bars ('|') is also a *Comma Expression* which represents that it may be any of the *Comma Expression* in the list. This kind of *Comma Expression* is called a *Selection Comma Expression*

- A list *Comma Expression* separated by commas (',') and enclosed by '(' and ')' is also a *Comma Expression*, which represents the aggregation of the *Comma Expression* enclosed. This kind of *Comma Expression* is called an *Aggregational Comma Expression*

The *Meta Path Expression* we mentioned above is similar to an ordinary path expression in ordinary object-oriented languages. It may or may not start with a scope name (meta type name) and dot ('.') is the separator along the path expression.

The BNF rules (in a format similar to the rule format of YACC programs) for a meta operation can be described as following:

```
<meta-operation>      : <TYPE-Name> "::"  <function-spec> ';'
                      | <function-spec> ';'
                      ;
<TYPE-Name>           : IDENTIFIER;
            /* it can be a pre-defined meta-type name; */
            /* or a new meta-type name of new components in the CQL; */
<function-spec>       : <function-name> '(' <parameter-list> ')'
                      ;
<function-name>       : IDENTIFIER;
            // usually it is some predefined operation name for
            // this meta-type.
<parameter-list>       : <parameter>
                       | <parameter-list> ',' <parameter>
                      ;
            // the parameter is the so-called "comma expression"
            // in our system.
<parameter>           : <comma-expression> <action>
<comma-expression>    : STRING
              // ordinary string def.,  treated as a meta token.
```

```
                    | <meta-path-expression>
                    | <option-expression>
             // represents the optional in the language spec.
                    | <list-expression>
             // represents the repeat components in the language spec.
                    | <selection-expression>
             // represents the select component in the lang. spec.
                    | '(' <parameter> ')'
             // represents the aggregation of the components in the lang.
                    ;
<meta-path-expression>
                    : <attribute-name>
                    | <TYPE-name> '.' <attribute-name-list>
                    ;
<attribute-name>    : IDENTIFIER;
              // usually the attr. in the meta type.
<attribute-name-list>
                    : <attribute-name>
                    | <attribute-name-list> '.' <attribute-name>
                    ;
<option-expression> : OPTION-BEGIN <parameter> OPTION-END
                    ;
          // OPTION-BEGIN is usually "["
          // OPTION-END is usually "]"
<list-expression>   : LIST-BEGIN <parameter> "," <list-separator> LIST-END
```

```
                // LIST-BEGIN is usually "("
                // LIST-END is usually ")"
                        | LIST-BEGIN <parameter> LIST-END
                // using the default list separator or terminator
                        ;
<list-separator>        : "s" STRING
                // which means the separator of the ele. is the string
                        | "e" STRING
                // which means the terminator of each ele. is the string
                        ;
<selection-expression>
                        : <parameter> SELECTION-SEP  <parameter>
                        | <selection-expression> SELECTION-SEP <parameter>
                // SELECTION-SEP usually is "|"
                        ;
```

5.3.2 Algorithms for Generating YACC Rules

The above BNF rules represent our basic ideas about meta operations and the *Comma Expression* for the parameters of the meta operations [4]. The algorithms for generating the BNF rules in YACC program format for the target language from the meta operations defined above are as follows:

(A) From a meta-operation definition, a top level BNF rule can be generated for this meta operation. The left hand side of this top level BNF rule is generated by

[4]Here for simplicity we omit the definitions of `<action>` and parameter naming mechanisms which are quite easy to process.

combining the meta type name (if none, use -GLOBAL as the default type) and the meta operation name of the meta-operation, and the right hand side consists of the elements generated by the parameters for the meta operation.

For example, from meta statement (2) in the above example, our system will generate the following BNF rules:

```
CLASS-create : "class" TYPE-IDENTIFIER
               CLASS-create-option1 "type" "tuple"
               "(" ATTRIBUTE-list ")" CLASS-create-option2
               "end"
             ;
CLASS-create-option1 : /* null */
                     | INHERITANCE
                     ;
ATTRIBUTE-list : ATTRIBUTE
               | ATTRIBUTE-list "," ATTRIBUTE
               ;
..., ...
```

The left hand side symbol for the top level syntax rule (the first rule) generated by our system is CLASS_create.

(B) Generating the right hand side elements from the parameters in the meta operation. Each parameter will generate a symbol with zero or more BNF rules for this symbol according to the following rules:

1. If the parameter is a STRING, the symbol generated is this STRING itself, and

no more BNF rule will be generated for this parameter; In the above example, `"class"`, `"type"`, ..., fall into this category. And in the generated YACC rules, `"class"`, `"type"` are the symbols for the corresponding parameters.

2. If the parameter is a meta-path-expression, we have to look at the domain of this meta path expression. There are several cases:

   (a) If the domain is TOKEN (TYPE.Name in statement (2) falls into this category because the domain of Name in meta type TYPE definition is TOKEN), our system will check to see if there is any further specification about the domain (in our example, the domain of Name is further specified as `TYPE-IDENTIFIER`). The symbol generated by this parameter will be the most specific TOKEN domain. Corresponding semantics checking actions may also be generated to check along the path as we mentioned in Chapter 3.

   (b) If the domain is another meta type, the symbol generated will be the symbol representing this meta type. In the above example, the domain of `Inheritance` is `INHERITANCE`, so the symbol in the second clause of `TYPE-create-option1` is `INHERITANCE`.

   (c) If the domain is a list of *Comma Expressions*, then the symbol will be generated according to the rules for the list which will be described later. The default separator or terminators will be inserted into the rules generated by the list parameters. In the above example, the domain of `Attributes` is a list of `ATTRIBUTE`, and so it falls into this category.

3. If the parameter is an optional-expression, we will generate a symbol for this option, which is the type name of the operation combined with the digit string representing the current number of option expressions in the type. For a meta

operation without a type, we just assume a -GLOBAL type. The generated symbol will represent the parameter in the original rules, and two more rules for the symbol will be generated as follows:

```
<option> :  /* null */
         | <symbol-generated-by-the-enclosed-parameter-recursively>
         ;
```

In the above example, `TYPE.Inheritance` falls into this category. The symbol generated for this option is `CLASS-create` followed by `option1`. Two more rules are also generated for this symbol, one is null, the other is the symbol generated by `TYPE.Inheritance`, which, according to the above rules, should be `INHERITANCE`.

4. If the parameter is a list-expression, first we have to check the domain of the element in the list, and generate a symbol for this domain (denoted as `<symbol-for-this-domain>` here). The symbol generated for the list is then `<symbol-for-this-domain>-list`; and two more rules will be generated for this symbol:

```
<symbol-for-this-domain>-list
          : <symbol-for-this-domain>
          | <symbol-for-this-domain>-list
            <symbol-for-this-domain>
          ;
```

According to the rules about the the list separator or terminator, we should also insert corresponding symbols in the above rules. Corresponding semantics action for the domain checking will also be attached to the rules.

In the above example, `Attributes` is the same as `list(ATTRIBUTE)`, so it falls into this category. First we generate the symbol `ATTRIBUTE-list`, and then two more rules are also generated for `ATTRIBUTE-list`.

5. If the parameter is a selection-expression, then similar to the optional-expression, a new symbol in the format of `<type-name>-selection-<number>` will be generated for this parameter, and one more rule for this symbol will also be generated:

```
<type-name>-selection-<number>  : <symbol-for-1st-selection>
                                | <symbol-for-2nd-selection>
                                      ...
                                ;
```

6. If the parameter is an aggregation of several comma expressions, we will generate a new symbol in the format of `<type-name>-aggreg-<number>`, and one more rule for this symbol will also be generated:

```
<type-name>-aggreg-<number>  : <symbol-for-1st-element>
                               <symbol-for-2nd-element> ...
                             ;
```

From the above BNF rule definitions for the meta operations and algorithms to generate BNF forms in the format of YACC rules from the meta operations, we can see that a YACC/LEX can be easily constructed to implement the algorithm.

5.3.3 Semantic Actions Hooked to the Syntax Rules

For each generated YACC rule, some semantic actions are also generated and attached to the corresponding part of the rule. The following is one of the above rules hooked with some semantic actions for this rule:[5]

```
CLASS-create :
               { /* allocate a new class entry in the symbol table, etc */
                 current-new-class-entry = malloc(sizeof(ClassTypeInfo));
                 ...
               }
                "class" "type" TYPE-IDENTIFIER
               { /* store the string value of this TYPE-IDENTIFIER */
                 /* in the class entry in the symbol table.         */
                  current-new-class-entry -> Name = $4;
                  ...
               }
                TYPE-create-option1
               { /* store the inheritance info. in the table */
                 ...
               }
               "type" "tuple" "(" ATTRIBUTE-list ")" TYPE-create-option2
                "end"
               { /* create a class in the meta schema */
```

[5]The following actions are based on our test implementation. We assume the variable **current-new-class-entry** points to the current entry of the class symbol table. **ClassTypeInfo** is a type defined for storing information of class definition. Please refer to the YACC manual for details.

```
                /* depends on the underlying OODBMS */
                ...,
            }
          ;
```

We can see that these actions are mainly dependent on the semantics of the meta operations that generate the rules. Since the basic semantics of each meta operation is predefined and well-known to our system, these semantic actions can be generated automatically by our system. In the above algorithm for generating of the YACC rules, we also mentioned how to generate some constraint checking semantic actions.

Language specifiers can also provide their own semantic actions to some meta operations or the *Comma Expressions* in the meta operations, and these actions can be mapped to the actual YACC semantic actions in a straight forward manner. A more formal system for the semantic action description is omitted in this thesis.

5.3.4 Parser Stack Design and Type Definition for the Symbols

In an YACC program, besides the syntax rules, we also have to define what kind of information we should keep track of for the symbols appearing in the syntax rules. This is done through the parser stack design and type definitions for the symbols. A parser stack is usually a union of the types defined for the symbols, and types for the symbols are decided according to our meta type definitions and what kind of information we should keep track of for the meta operations for the meta types. For example, for the symbol generated from the meta constructor for the meta type

INHERITANCE, we have to record all the information about INHERITANCE. Hence, we have the following definition in our YACC program:

```
%type<inheritanceInfo> INHERITANCE
```

where `inheritanceInfo` is a member of the union for the parser stack. The type of `inheritanceInfo` is `InheritanceInfo` which is actually some kind of direct translation from our meta type definition for INHERITANCE, i.e. [6]

```
InheritanceInfo {
    os_List<String*>        *SuperTypes;
    os_List<RENAME*>        *Renames;
}
```

We actually have predefined elements in the parser stack and type definitions for all the predefined meta types and meta operations in this way. For the new components constructed by the language specifiers, we can define the parser stack and types for the symbols in a way similar to that of the predefined meta types. This is because the new components are built on the predefined meta types (domains) and operations. Detailed description is omitted here.

[6]Here, we use Objectstore collection constructors for our test implementation, but actually, we can use simple C and C++ structure to represent this.

Chapter 6

Conclusions and Further Research

6.1 Summary

The rapid development in OODBMS and query languages for OODBMS during the past few years has presented two important problems to researchers and industry. One is the inter-operability between different systems in the multidatabase system environment. Even for the relational database systems, the problem of inter-operability is also a very important one. A lot of front end systems between the systems from different RDBMS vendors have come out and some standard connection protocols such as ODBC (Open DataBase Connection, a database inter-operability standard by Microsoft) have been proposed and developed. Since there are many more variants of OODBMS's and all of them are much more complicated than RDBMS's, we believe that the inter-operability between different OODBMS's will be more important to the users.

Another problem is the development of Customized Query Languages. Because of the complexity of OODBMS and the variants of OODBMS applications, CQL's are more suitable interfaces for ordinary end users of many OODBMS applications. Generally, development of a moderate complex CQL directly based on the basic services provided by a general OODBMS needs a lot of expertise about the underlying OODBMS, formal language specification, compiler techniques, query processing and optimization. The development also takes a lot of time and effort. Hence the development tool for CQL's based on OODBMS is very important for OODBMS application developers.

In this thesis, we proposed a new approach for OODBMS query language front end development which can help systematically solve the above two problems together. We suggested and also provided a concrete design of a general framework for query language front end specification based on an abstract Object Model and abstract query language. A lot of different CQL's and query languages of different systems can be specified based on our systems. A specification based on our system will be object-oriented, using message passing instead of a formal grammar specification. Language specifiers can reuse a lot of common components pre-built in our system, dramatically reducing the work of specification. Most of the specification itself is also the "implementation" in the sense that our system understands the syntax as well as the semantics of the specification, hence a query processor for the specified query language can be directly generated from the specification. So the language specifiers do not have to know very much about formal language specification, compiler techniques, query processing. We can also generate several query processors based on several different underlying OODBMS's for the specified query language so that the query based on the specified query language can actually run on different underlying platforms. At the same time the query language developers may not have to understand all the underlying OODBMS's at an expert level.

Our work combined the techniques and research results from several different areas. First one is the rapid development of OODBMS's. Most of basic concepts and functionalities of OODBMS are accepted widely, and object models and query languages have come to a mature stage. Combining the features from the most important systems and standards, we proposed a general abstract object model, and this model is the corner-stone of our whole system.

Another important technique used in our system is the formal language specification, compiler techniques and automatic program generation. Since our system is a tool for query language specification, we have to consider all the basic aspects of formal language specification and try to make the task of language specification easy and convenient for ordinary developers. LEX and YACC are two basic general tools for formal language specification and automatic compiler generation. The concrete design and implementation of our system is actually based on these two tools, although in the user interface, we use some quite different formats. Therefore the concrete design of our system is heavily influenced by YACC and LEX, and the implementation of our system is also based on them (but the basic design idea is not dependent on these tools. Our design should also be able to be implemented by other compiler-compiler tools).

The most important feature of our system, also one of the main contributions of our research is that we used an object-oriented methodology in the design of the whole system, combining the above two main kinds of techniques in a seamless way. The interface provided to the language specifiers is also in object-oriented way. We abstracted the common components from different systems and used object oriented way to analyze the abstract components. So the abstract object model is pre-built in our system in an object-oriented manner. Object-oriented style is also used in the formal language specification. Syntax appearance and the semantics aspect of the formal language become different facets of the abstract objects in our system, and the syntax specification usually also serves as the semantics specification of the same components. To our knowledge, our work is the first one to use object-oriented methodology in formal language design (including specification and implementation).

Our object-oriented design and presentation of the interface of our system also reflects the object-orientation of the design and the implementation of our system. Actually, some of the basic meta-types and objects have the corresponding internal meta-types and almost all the meta-operations defined on the abstract objects have the corresponding internal meta-operations. For example, the meta-type TYPE presented to the language specifiers has a corresponding meta-type "type" in our implementation. All the meta-operations of TYPE also have the internal representation, like the meta-operation `TYPE::create` presented to the language specifiers has the internal corresponding meta operation type::create(...), which will be mapped onto any actual underlying OODBMS in which the generated query processor will run. These internal representations are defined in an object-oriented way and are some general abstractions of meta-types and operations of different underlying OODBMS's. When the language specifiers want to generate the query processor running on a certain underlying platform, the general abstraction of these meta types/objects and operations will then be mapped onto the actual types/objects and operations of that underlying OODBMS. So our meta system serves as a shield between the front end query languages and the underlying database systems.

6.2 Future Research

There are some area of research that can be done in the near future along the direction of this thesis.

- **Prototype Implementation:** Although we have partially implemented our system based on Objectstore, there is still some more work to do for a prototype system of our design. In our current design, we had to ignore some features of query languages, such as virtual attributes specification, transactions management, etc. Also, our design of the whole system is still not very exquisite. For a more realistic prototype, a more careful and comprehensive design is needed.

  Our current testing experiment is only based on Objectstore. That is to say, we only generate different query front ends (query processors for different query languages) on Objectstore. Although, according to our studies, generating query processors for other OODBMS's from the query language specifications based on our system should be quite similar to our current test, more complete inter-operability between query languages of several different OODBMS can only be implemented if we implemented our system on several other popular OODBMS's, such as O2, Objectivity, etc. So prototypes based on other OODBMS's should also be constructed.

- **Graphic User Interface for Query Language Specification:** Although we used object-oriented style in our interface to the language specifiers, and the language specification in our system is easier than that in ordinary formal grammar, a more friendly graphic user interface can be designed based on our abstract meta components. In the graphic user interface, all the predefined meta-types/objects and meta-operations can be displayed to the language specifiers. Customization and tailoring can be done directly by changing the corresponding parts on the meta-operations displayed in the GUI. Through this kind of graphic user interface, the language specifiers can have a better global understanding of our system and they can use our system more efficiently.

- **Query Optimization:** In our thesis, we have not considered query optimization strategies in detail. But query optimization is one of the crucial parts of query processing. Although a lot of query optimization strategies for traditional SQL may still be applicable to the query languages of OODBMS's, there are a lot of more complex situations where we have to consider some new optimization algorithms. Since current optimization algorithms are mainly based on the predefined operations for the predefined types in the system, while in object-oriented query languages a lot of user defined operations are involved, some optimization hints provided by the designers based on the operation specification/implementation should be very useful to the system. Rules (operation algebraic rules, logic rules or code segment transformation rules) will be some kinds of vehicles for language specifiers to provide these hints. Because of the complexity of the object manipulation language, optimization algorithms used in compiler systems for ordinary program languages may also be very useful for the complicated query languages of OODBMS's. This area is a kind of inter-discipline of OODBMS and programming languages. A lot of research is needed in this area.

- **OO Design of Generic Language Specification:** Formal language specification is very important in a lot of application areas. And traditional formal language specification is too complicated for ordinary developers. Our thesis proposed a new way for formal language specification, especially for the formal specification of a language family based on some similar semantic models. An abstract semantic model and a default language (or a language with abstract syntax) can be predefined in an object-oriented way. Each language in the language family will be one kind of concrete appearance of the abstract language. With the help of compiler techniques and automatic program generation, the languages can be implemented by a predefined system for this language family. For example, in the telephone network, different digital switches may be used, and different switches from different manufactures use different sets of switch commands. But the underlying semantics of these switch commands are quite similar. We believe that our design idea can be used for the inter-operation of these different switch commands.

Bibliography

[1] E. Bertino, M. Negri, G. Pelagatti, L. Sbattella, "Object-Oriented Query Languages: the Notion and the Issues", *IEEE Trans. on Knowledge and Data Engineering, Vol.4, No.3*, June 1992.

[2] P. Butterworth, A. Otis, J. Stein, "The Gemstone Object Database Management System", *Communications of the ACM, Vol.34, No.10*, October 1991.

[3] R.G.G. Cattell et al, "The Object Database Standard: ODMG-93", *Morgan Kaufmann Publishers*, August 1, 1993.

[4] R.G.G. Cattell, "Object Data Management" *Addison-Wesley Publishing Company*, 1991.

[5] O. Deux et al, "The O_2 Systems", *Communications of the ACM, Vol.34, No.10*, October 1991.

[6] K. Kulkarni, A. Eisenberg, "Evolution of the SQL Standare: SQL2 and SQL3", *SIGMOD'92 Tutorial: Slides*, June 3, 1992.

[7] K. Kulkarni, A. Eisenberg, "Introduction to SQL3", *SIGMOD'92 Tutorial: Slides*, June 3, 1992.

[8] W. Kim, "A Model of Queries for Object-Oriented Databases" *Proc. of the 5th Intel. Conf. on VLDB*, 1989.

[9] W. Kim, "Object-Oriented Databases: Definition and Research Directions", *IEEE Trans. on Knowledge and Data Engineering, Vol.2, No.3*, September 1990.

[10] G. M. Lohman, B. Lindsay, H. Pirahesh, K. B. Schiefer, "Extensions to Starburst: Objects, Types, Functions, and Rules", *Communications of the ACM, Vol.34, No.10*, October 1991.

[11] W.S. Luk, A.Y.L. Choi "Dynamic Spatial Query Language: A Customized Query Language for Object-Oriented Database Systems" *IEEE COMPSAC*, Tokyo, September, 1991.

[12] J. Melton(Editor), "(ISO/ANSI) Working Draft Database Language SQL(SQL3)" *ISO, ANSI*, July, 1992.

[13] V. M. Markowitz, A. Shoshani, "Object Queries over Relational Databases: Language, Implementation, and Applications" *IEEE Data Engineering*, 1993.

[14] W. Meng, C. Yu, W. Kim, et al, "Construction of a Relational Front-end for Object-Oriented Database Systems." *IEEE Data Engineering*, 1993.

[15] P. Oosterom, T. Vijlbrief, "Building a GIS on Top of the Open DBMS "Postgres"", *EGIS'91, Brussels, Belgium*, 1991.

[16] Persistence Software, Inc. "Persistence^{TM} User Manual Version: 1.2" *Persistence Softeare, Inc.* March 1993.

[17] The Committee for Advanced DBMS Function, "Third-Generation Database System Manifesto" *Sigmod Record, Vol.19, No.3*, September 1990.

[18] M. Stonebraker, G. Kemnitz, "The Postgres Next Generation Database Management System", *Communications of the ACM, Vol.34, No.10*, October 1991.

[19] R. Snodgrass, "The Temporal Query Language TQuel" *ACM Trans. on Database Systems, Vol.12, No.2*, June 1987.

[20] T. Vijlbrief, P. Oosterom, "The GEO++ System :An Extensible GIS", 1992.

[21] Sun Microsystems, Inc., "Lex- A Lexical Analyzer Generator", Programming Utilities for the Sun Workstation.

[22] Sun Microsystems, Inc., "Yacc- Yet Another Compiler Compiler", Programming Utilities for the Sun Workstation.