OBJECT COMPREHENSION TRANSLATION FOR OBJECT ORIENTED DATABASES

ALEXANDER LAKHER

A MAJOR REPORT
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

APRIL 1997

0-612-47733-9

Abstract

Object Comprehension Translation for Object Oriented Databases

Alexander Lakher

Object Comprehensions are a new query notation introduced in 1994 by Chan and Trinder to provide a declarative query language for object-oriented databases. Object Comprehensions allow queries to be expressed clearly, concisely, and processed efficiently, while incorporating many features that are missing from or inadequate in existing object-oriented query languages such as support of object-orientation, computational power and support of collection. However there is no object-oriented database (OOD) so far which incorporates Object Comprehension Language (OCL) as a query interface. This paper introduces an experimental translator to translate OCL into the O++ query language for Ode, an object-oriented database system from AT&T.

Acknowledgements

I would like to thank my supervisor. Dr. Gregory Butler. for his patience and valuable guidance.

I am also grateful to Georges Ayoub for his excellent team work and an inexhaustible sense of humor.

My special words of gratitude go to my parents and brother for their unceasing encouragement. It is my mother who taught me how to look at the line between success and failure as a blurred one.

Contents

List of Tables

List of Figures

Chapter 1

Background

1.1 Introduction

There are many object-oriented query languages [2], [7] that have been proposed in recent years. Some of them are designed particularly for object-oriented databases (OOD) and some adapted from other areas: the relational data model and its extension [9], object-oriented programming languages [5]. According to [6] all of them are inadequate in one way or another. These inadequacies are categorised in [4] into four groups:

- *Support of object-orientation.* A few OO query languages do not capture the class hierarchy which is defined by the *ISA* relationship between classes defined in a database schema.

- *Structuring power.* It refers to the ability to explore and synthesize complex objects that are the components of OOD. The creation of a new object may require a collection of objects as a parameter. To do so a query language must provide something like nested queries and allow orthogonal composition of constructs.

- *Computational power.* Recursion and quantification are two examples that characterise the computational power of a query language. Traversal recursive queries as well as quantification are supported poorly. Recursive queries with computation are supported even worse.

- *Support of collection.* Usually "Set" is widely supported and its operations are well defined. It is not the case for other collection classes; interaction between different collection classes is unclear.

Pointing out the above-mentioned inadequacies [4] introduces a new query notation called *Object Comprehensions*, which takes into consideration the fundamental properties of object-oriented data models and eliminates above-named drawbacks. Object Comprehensions Language (OCL) is based on *List Comprehensions* in [10] which is clear, consize and powerful. The extention of List Comprehensions to Object Comprehensions was done by consolidating and improving constructs found in existing query languages.

1.1.1 Overview of the OCL project components

This report is a part of a team project work, the main idea of which is to test OCL, the new powerful query notation. For that purpose OCL queries are to be translated into O++, an interface language of the object oriented database Ode. Translated O++ queries are to be run against a university database based on Ode. This idea initialized a team work which consisted of three interrelated projects. First of all the required database system is to be built and managed. This constitutes one of components, the database project. Building a database system in turn requires a symbol table for such a database, which describes a plot for another component, the schema project. This project is responsible for the implementation of the above mentioned symbol table. The third team work component is to provide the translation of user specified OCL queries into an interface query language that could be recognized by and run against the target database created in the database project. Now we can have a closer look at each of these components and their mutual relationships.

The team work consists of the following three interrelated components:

- *Creating an Ode-based university database as a target for the translation.*
  The essence of this project is to set up and manage the University sample database. It is decided that the target database is to be based on Ode, experimental object-oriented database system from AT&T. It is defined, queried and manipulated in O++, the database interface programming language which is

based on C++ programming language. The University sample database reflects a university model, a schema of interrelated objects of O++ classes such as Department, Staff, Student, Course, etc. The O++ based queries are to be run against the University sample database. The results are obtained by utilizing the Ode built-in query facilities and implemented O++ classes to support OCL concepts, so called data structure utilities for query processing such as Set, List and Bag. The query results then would be printed in the form of collection of solutions.

- *Implementation of a university database symbol table.*
  The purpose of the schema project is to implement a symbol table for the above mentioned University sample database of the database project. It is to look into BNF grammar for the schema definitions and implement the facility to input views described as schema's for those OCL queries which create objects of new classes .

- *Implementation of the OCL-to-O++ translator.*
  Presented in this report this project is focusing on the translator implementation and translation of the OCL queries into O++, an interface language for the Ode database system. Those OCL-to-O++ translated queries are to be run against the University sample database built in the database project. The scope of the translation project is set to design and implement experimental OCL translator. The issues of translator's code optimization, error handling, user interface as well as the topic of Object Comprehension queries optimization are not in the scope of this project.

1.1.2 Organization

The organization of this report is as follows. The remaining sections of this chapter provide a background of Comprehensions and an overview of Ode. Chapter 2 describes the sample data model, OCL and sample queries. Chapter 3 presents the desired translations of the OCL queries. Chapter 4 outlines the implementation of the translator. Chapter 5 concludes. Bibliography is followed by appendices which contain the actual code of the translator.

1.2 Comprehensions

1.2.1 Set Comprehensions

The inspiration for *comprehensions* was the standard and convenient mathematical notations for sets. For example in mathematics the set of squares of all the even numbers in a set S would be written as:

$$\{x^2 | x \in S \wedge even(x)\}$$

Comprehensions first appeared as *Set Comprehensions* in an early version of the programming language NPL that later evolved into *Hope* [11] but without comprehensions. They were followed by *List Comprehensions* [12].

1.2.2 List Comprehensions

A full description of *List Comprehensions* can be found in [10]. The above mathematical expression written using *List Comprehensions* would have a look:

$$\{x^2 | x \leftarrow L; even(x)\}$$

where L stands for a list.

List Comprehensions have been incorporated into several functional languages. e.g. Miranda [13] and Haskell [8].

The syntax of list comprehensions is as follows. where E stands for an expression. Q stands for a qualifier, P stands for a pattern, and O stands for an empty qualifier:
E ::= [E | Q]
Q ::= E | P ← E | O | Q ; Q

The result of evaluating the comprehension [E | Q] is a new list, computed from one or more existing lists. The elements of the new list are determined by repeatedly evaluating E, as controlled by the qualifier Q. A qualifier is either a filter or a generator, or a sequence of these. A filter is a boolean-valued expression, expressing a condition that must be satisfied for an element to be included in the result. An example of a filter in the example above is *even(x)*. A generator of the form T ← E, where E is a list-valued expression, makes the variable T range over the elements of the list. An example of a generator is $x \leftarrow L$ above. More generally, a generator of the form P $\leftarrow$ E contains a pattern P that binds one or more new variables to components of each element of the list.

1.2.3 Object Comprehensions

The generalization of list comprehensions has brought *collection comprehensions*, which provide a uniform and extensible notation for expressing and optimizing queries over many collection classes including sets, lists, bags, trees and ordered sets. That is what in [4] is called *object comprehensions*. The most significant benefit is that, although each primitive operation will require a separate definition for each collection class, only one query notation is needed for all these collection classes. A single definition is all that is required for high-level operations defined in terms of collection comprehensions. It significally reduces the syntactic complexity of the query notation:

$$\{x^2 | x \leftarrow C; even(x)\}$$

where C stands for a collection, i.e. could be a bag, a set or a list. Here are some examples of object comprehensions using the notation suggested in [4].

Example 1. Return a set of elements of the bag B. A bag allows duplicates however they are to be eliminated in the resulting set.

```
Set [ e <- B | e ]
```

Example 2. Return a bag of elements of the list L. Possible duplicates are preserved however the order of the elements is lost.

```
Bag [ e <- L | e ]
```

Example 3. Return a list of elements of the list L provided they comply with some specific 'condition'.

```
List [ e <- L ; condition(e) | e ]
```

It is worth mentioning that comprehensions are a declarative specification of a query, and as it is shown in [4], are a good query notation for being concise, clear, expressive and easily optimized. However the optimization was out of scope of this project and left for the future work.

1.3 Overview of Ode and O++

Ode is an object-oriented database based on the C++ object model. Ode is the product of AT&T Bell Laboratories. The programming interface to Ode is the O++ database programming language. O++ extends C++ with facilities for creating and manipulating persisting objects, organizing persistent objects into clusters, defining and manipulating sets, querying the database, specifying constraints and triggers, and running transactions.

The Ode database is based on a client-server architecture. Each application runs as a client of Ode database. Multiple Ode applications running as clients of the database server can concurrently access the database. Ode also supports single user applications that can run without a separate server.

Ode 4.0, the current release, consists of the O++ compiler (OO) and the object manager library. Database applications are written on O++. The O++ compiler translates an O++ program into a C++ code, which contains calls to the Ode object manager library. The library provides facilities for creating and manipulating persistent objects. The translated program is then compiled with the C++ compiler and linked with the object manager library to form an executable program as indicated in Figure 1.

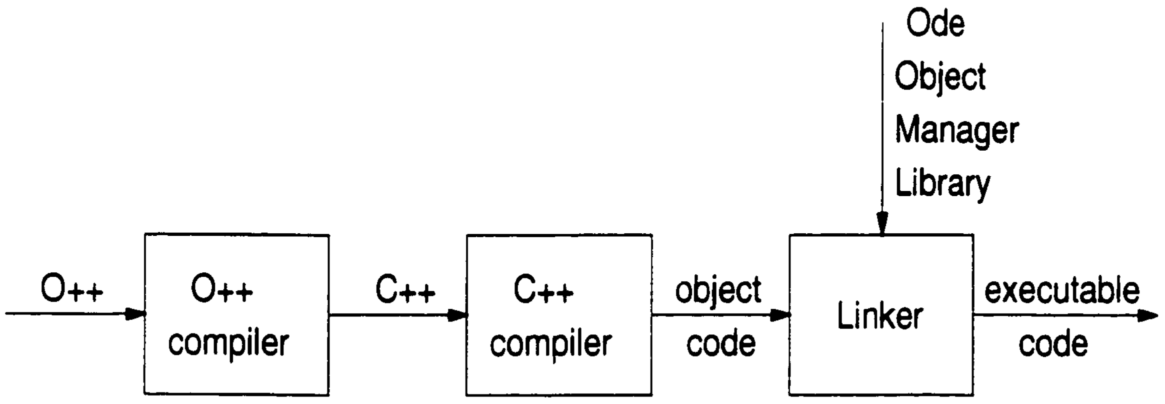

Figure 1: Compilation of an O++ program.

O++ has the following features to facilitate a database access. Objects of class types can be accessed by using the associative *for* loop as illustrated:

```
for ( s in STUDENTS )
    { ... }
```

This loop will iterate over all the *s* of type *STUDENTS* in the default database.

Another important feature is *suchthat* clause. It can be used to restrict the search to objects that satisfy a boolean expression, e.g.:

```
for ( s in STUDENTS )
   suchthat (s.age > 125)
   { ... }
```

We will use these two basic constructs to express our queries in O++. Here is the the concrete query example.

Example.

```
Set<STUDENTS> tempSet;    //result collection declaration: set, bag or list
for ( s in STUDENTS )                                  //'for' loop iteration
   suchthat(s.address.city == "Montreal")              //restriction applied
   tempSet.add(s);                                     //result is ready
```

For further O++ query examples please see Chapter 3. They are presented there as the translations of OCL sample queries from Chapter 2.

1.3.1 Objects in O++

The O++ object facility is based on the C++ object facility and is called the *class*. *Volatile* objects are allocated in volatile memory and are the same as those created in ordinary programs. *Persistent* objects are allocated in persistent memory and they continue to exist after the program creating them has terminated, that is objects may exist between invocations of programs. In O++ persistence is a property of the object instance, not the class. Objects of any class can be persistent. Some objects of a class can be persistent, while others are volatile. In O++ the *pnew* operator creates a new persistent object and is otherwise identical to *new* operator in C++ which is to create a new volatile object. The *pdelete* operator deletes a persistent object. Each persistent object is identified by a unique identifier, called the object identity (oid). The object identity is referred to as a *pointer to a persistent object.* O++ provides facilities for associating *constraints* with an object. These are specified as part of a class definition, and are treated as *members* of the class. The specified constraint conditions are checked every time an instance of that class is updated (through a public member function), a new instance is created, or an old one is removed. If the constraint is found to be violated, the constraint 'fires', executing the action part associated with it, if any. After the action part is executed, the constraint is checked again. If it is still not satisfied, then the transaction attempting the update causing the constraint violation is aborted, and all its updates are undone.

The constraint facility provided in O++ is *intra-object* in that when an object is updated only the constraints associated with it, through its class definition, are checked. This restriction is for the reasons of efficiency, as well as in accordance with the spirit of localized processing of object-oriented programming. It is not practical to check every constraint with every object, every time that any update is made in the system. Note that there is no restriction on referencing or even modifying other objects in the condition or action part of a constraint. Constraints can be *hard* or *soft.* Hard ones are checked as soon as the object is updated, and must be satisfied immediately. Soft constraints checking is deferred until the end of the transaction causing the update. Inter-object constraints almost always must be soft since the constraint may be violated after one object has been updated, but before the other one has.

1.3.2 Transactions in O++

Transactions access and manipulate the objects in the database by invoking operations on the objects. They also invoke transaction management primitives. If a transaction has invoked operations on an object but has not yet committed, then this transaction is *responsible* for the uncommitted operations.

Each O++ source file that uses O++ database facilities must include *ode.h* file. Class *database* is automatically available by including this header. This class provides functions for manipulating (such as closing, opening, etc.) the database and naming persistent objects. Only the following database operations

- **open**: opens the database identified by **name**. On successful completion, **open** returns the pointer to the **database** object; on failure returns NULL.
- **close**: is used to close a database previously opened by calling **database::open**. Returns zero on success and non-zero on failure.
- **remove**: deletes a database previously opened by calling **database::open**. Returns zero on success and non-zero on failure.

can be invoked from outside a transaction body. All other operations must be invoked within the transaction body. Thus all code interacting with the database, except database opening and closing, must be within a transaction block. By default Ode uses 2-version 2-phase (2V2P) locking which allows multiple transactions to read an object and one transaction to write an object.

There are three kind of transactions: update, read only, and hypothetical. Update transactions have the form:

```
trans { ... }
```

Read-only transactions have the form:

```
readonly trans { ... }
```

Hypothetical transactions allow users to pose 'what-if' scenarios and have the form:

```
hypothetical trans { ... }
```

Here is an illustration of a transaction block in a database called **coffee**:

```
#include <ode.h>
...
main()
{
   database *db;
   ...
```

```
    if ((db = database::open("coffee")) == NULL)
       error("no coffee for today - cannot open database");
    trans {
          ...
          }
    db->close();
}
```

Transactions are aborted using the `tabort` statement. The macro `old(X)` can be used within a transaction to return the value of X at the beginning of the transaction, where X is a persistent object. Similarly the macro `changed(X)` returns TRUE if X has been modified from `old(X)` within the course of the current transaction, and FALSE otherwise.

1.3.3 Events

Events can be *basic* and *composite*. The latter is composed from basic ones and other composite events.

Basic Events

There are three sorts of basic events.

- *member function call events*: invoking public member functions on a persistent object causes previously declared events to be posted to the object, e.g. the call `pObject->function1()` where `pObject` is a persistent pointer and `function1` is a public member function causes the event **`before function1`** to be posted to this object before the call is executed and the event **`after function1`** after its execution.

- *transaction events*: the run-time system posts two types of transaction events. 1. **`before tcomplete`** posted just before completion of the transaction which happens just before a transaction attempts to commit. Conceptually, after the last action in a transaction unit has completed execution, **`before tcomplete`** is posted to each object that was accessed by the transaction and is interested in this event.

2. `before tabort`: posted just before a transaction is about to be aborted. It is posted to each object that was accessed by the transaction and is interested in this event. The posting occurs immediately prior to the system rolling back transaction actions in response to encountering a `tabort`, transaction abort command. System aborts do not cause any events to be posted.

- *user-defined events*: users can define new events; such events must be posted explicitly. For example, the event declaration '`event gateClosing;`' defines a new event which is not associated with any method. User-defined events are posted to a persistent object by calling the function `PostEvent`. For example, `PostEvent(p, gateClosing);`
posts the event `gateClosing` to the object referenced by the persistent pointer p.

Composite Events

Composite events are composed from basic events using event composition operators.

- *Sequence Operator*: denoted by comma, parentheses must be used to delimit the sequence expression, e.g. `(after gateOpening, before gateClosing)`.

- *Disjunction Operator*: operator `||` is used to specify the occurrence of one of two events, e.g. `after gateOpen || before gateClosing`.

- *Conjunction Operator*: Event `A && B` is satisfied if both A and B are satisfied simultaneously by the posting the same basic event. If A and B are both distinct basic events, then the above composite event will never be satisfied.

- *Negation Operator*: There are three operators used to specify an event that does not satisfy the specified pattern.
1. The ~ operator. Event `~before gateClosing` is satisfied by any basic event other then `before gateClosing`.
2. The ! operator. Event `!A` is satisfied by any sequence of events that do not satisfy A.
3. The - operator. Event `A - B` is satisfied by any sequence of events that satisfies A but not B.

- *Repetition Operators*: There are two repetition operators.

  1. The `*` operator. Event `*A` is satisfied by a sequence of zero or more occurrences of A events.

  2. The `+` operator. Event `+A` is satisfied by a sequence of one or more occurrences of A events.

- *Count Repetition Operators*: There are three count-repetition operators.

  1. The `relative` operator: Event `relative(3, after f)` specifies a composite event which is satisfied by the third and succeeding occurrences of the event `after f`.

  2. The `every` operator: Event `every(3, after f)` is satisfied by the third. the sixth. etc. occurrence of event `after f`.

  3. The `choose` operator: Event `choose(3, after f)`— is satisfied just after the third occurrence of event `after f`.

- *Event Ordering Operators*: Operator `relative` is overloaded. Event `relative(A, B)` is satisfied by every occurrence of a B event that begins and completes after an A event has occurred. Events A and B cannot overlap in time. Operator `prior` is like `relative` plus it allows overlaping.

- *Masking Events*: A mask is a condition applied to an event to determine if the event is of interest. A mask can be an arbitrary side-effect free O++ expression. e.g. `after gateClosing & (time > 10pm)`.

- *Pipe Operator*: Denoted as `|`. it is used to filter out uninterested events. e.g. `(b1 || b2 || ... || bN) | E`.

Events are inherited by derived classes. All basic events must be explicitly declared in the following form:

```
event e1, e2, ..., eN;
```

where the *es* are basic events.

1.3.4 Triggers

Triggers monitor the database for some conditions. When these conditions become true the associated trigger action is executed. Triggers are associated with objects.

There are two types of triggers: *once-only* (default) and *perpetual* (specified using the keyword **perpetual**. A once-only trigger is automatically deactivated after the trigger has fired and must be reactivated explicitly if desired. A perpetual one is automatically reactivated after being fired. Firing means that the action associated with the trigger is scheduled for action. Triggers are specified within class definitions. They are set by explicitly activating them after the object has been created. A trigger T1 associated with an object whose *id* is objectID is activated by the call

```
objectID->T1(arguments)
```

The trigger activation returns a trigger id (value of predefined class **TriggerId**) if successful; otherwise it returns a 0.

Triggers may be deactivated explicitly before they have fired:

```
~trigger-id
~object-id->T(arguments)
```

Chapter 2

OCL

2.1 The Sample Data Model

The sample data model is a simplified university system which records information about students. staff members of a university, its academic departments and courses. Figure 2 presents the data model.

The class *Person* has two subclasses: *Student* and *Staff*: *Visiting Staff* is a subclass of *Staff*. *Tutor* inherits from both *Student* and *Staff* to incorporate students doing part-time teaching. The calculation of the salary of a tutor is different from that of a staff member. The variation is captured by overloading *Salary* method to *Tutor*. Every person has an address which is an object of the class *Address*. A student has at least one superviser. This is modelled by the *SupervisedBy* method as a list of staff members. Every and each student as well as a staff member is associated to a department of class *Department* by means of *Major* and *department* accordingly. Courses of class *Course* are *runBy* a department provided there is a staff member who *teaches* this particular course which enables a student to take this course via *takes*. A course may have a set of *prerequisites*. The schema definition is as follows:

```
Class Person isa  Entity
 methods
 name          : -->  String ,
 address       : -->  Address.
```

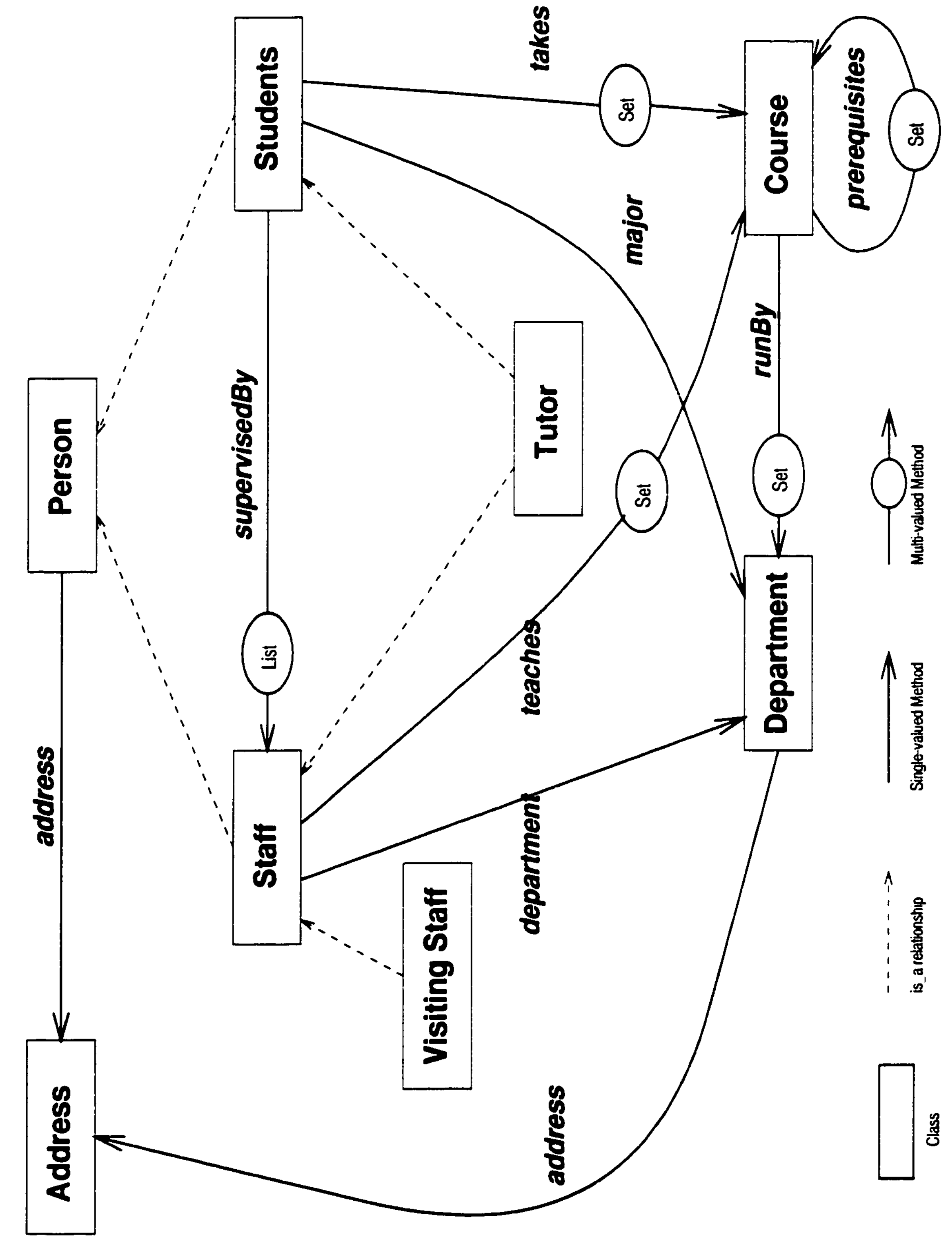

Figure 2: University Model Diagram.

```
Class Staff isa  Person
 methods
 department  : -->  Department,
 teaches     : -->  Set of Course,
 salary      : -->  Integer.

Class Student isa  Person
 methods
 major        : -->  Department,
 takes        : -->  Set of Course,
 supervisedBy : -->  List of Staff.

Class Tutor isa  Student, Staff
 methods
 salary       : -->  Integer.

Class Course isa  Entity
 method
 code         : -->  String,
 runBy        : -->  Set of Department,
 prerequisites: -->  Set of Course,
 assessments  : -->  Bag of Integer,
 credits      : -->  Integer.

Class VisitingStaff isa  Staff.

Class Address isa  Entity
 methods
 street       : -->  String,
 city         : -->  String.

Database is
  Persons     :  Set of Person,
  Departments :  Set of Department,
Courses      :  Set of Course,
StaffMembers:  Set of Staff,
Students     :  Set of Student,
Tutors       :  Set of Tutor.
```

Using this model the next section describes OCL by presenting different kinds of queries in OCL notation.

2.2 The OCL sample queries

Before starting the query examples we would like to elaborate on the grammar of object comprehensions. That would be useful while examining the queries and also later when discussing the implementation issues of the translator in Chapter 4.

Table 1: The OCL Grammar.

```
expression    ::= pathname
                  | {collectionType "[" qualifiers "|" expression "]"}
collectionType::= "Set" | "Bag" | "List"
qualifiers    ::= qualifier [ qualifiers ]
qualifier     ::= [ generator ] [ ; localdef ] [ ; filter ]
generator     ::= identifier "<-" pathname
localdef      ::= identifier "AS" pathname
filter        ::= pathname { op ( pathname | integer ) }
pathname      ::= identifier [ "." pathname ]
op            ::= ">" | "<" | "<=" | ">=" | "==" | "!="
digit         ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
integer       ::= digit [integer]
identifier    ::= letter [identifier]
letter        ::= a | A | b | B | c | C | d | D | e | E |
                  f | F | g | G | h | H | i | I | j | J |
                  k | K | l | L | m | M | n | N | o | p | P |
                  q | Q | r | R | s | S | t | T | u | U |
                  v | V | w | W | x | X | y | Y | z | Z |
```

Presented in conventional BNF it still might need some explanation. The best way is to do it by examples. The example of a typical *pathname* that is a compound indentifier is `s.address.city`. A typical *filter* is `s.address.city == "Vienna"`. A *generator* example: `s` $\leftarrow$ `Students`. A *localdef* example: `a AS s.address.street`. A query then would be expressed in the following style:

```
Set [ s <- Students ; a AS s.address.city ; a == "Vienna" | s ]
```

2.2.1 OCL sample queries

The following novel features of OCL as a query language should be noted:

- a predicate-based optimizable language providing support for the class hierarchy;
- numerical quantifiers for dealing with occurrences of collection elements;
- operations addressing collection elements by position and order;
- a high-level support for interaction between different collection kinds;
- recursive queries with computation.

Now we are to demonstrate object comprehensions by using queries posed against the described university database. Queries Q1 - Q6 demonstrate the support of Object-Orientation; Q7 and Q8 explore the result expression. Q9 - Q11 focus on generators. Quantifiers are highlighted in Q12 - Q17. Support of Collection is emphasized in Q18 - Q27. The last query, Q28, is to reflect the ability of Query Functions and Recursion. All queries demonstrate the declarative nature of object comprehensions as a query notation.

Each query presented in the following format:

$\boxed{\text{Query \#. The text of a query}}$

OCL version of a query

Method Calling and Dynamic Binding.

Encapsulation protects attributes of an object from being accessed directly. An access is to be made via a method. In Q1 *s.salary* represents the calling of method *salary* on a staff member object *s* drawn from StaffMembers. A method could be overloaded as it as the case with *tutor* whose salary is calculated in a different way.

$\boxed{\text{Q1. Return staff members earning more than \$1000 a month.}}$

```
Set [ s <- StaffMembers; s.salary > 1000 | s ]
```

Complex Objects and Path Expressions.

Support of complex objects implies that a method call may return an object which can, in turn, receive another method call and so on. Such a sequence of method calls is usually referred to as a path expression.

$\boxed{\text{Q2. Return tutors living in Glasgow.}}$

```
Set [ t <- Tutors; t.address.city = "Glasgow" | t ]
```

Object Identity.

In object-oriented data models, objects are represented by object identifiers which are essential for object sharing and representing cyclic relationships. Equality between objects is defined by the equality between their object identifiers.

$\boxed{\text{Q3. Return tutors working and studying in the same department.}}$

```
Set [ t <- Tutors; t.department = t.major | t ]
```

Class Hierarchy.

StaffMembers contains only members of the faculty. The only collection in the database that contains all visiting staff members is *Persons*. The elements of *Persons* can be of class *Persons* or its subclasses. In Q4 *hastype* returns true if person object p is an instance of class *VisitingStaff*.

$\boxed{\text{Q4. Return all visiting staff members in the university.}}$

```
Set [ p <- Persons; p HASTYPE VisitingStaff | p ]
```

In the following query the method *salary* is defined for visiting staff members but not for persons in general. The filter is applied only if the object is of specified class.

Q5. Return visiting staff members earning more than 1000 a month.

```
Set [ p <- Persons; p HASTYPE VisitingStaff WITH p.salary > 1000 | p ]
```

Local Definitions.

Local definitions simplify queries by providing symbolic names to expressions. They are particularly useful when a path expression is used in more than one place.

Q6. Return students whose major departments are in either Hill st. or U ave.

```
Set [ s <- Students; a AS s.major.address.street; a == "Hillst"
        OR  a == "Uave" | s ]
```

The Result Expression.

Operation that create new objects should be allowed in the result expression. Method *new* takes to parameters and creates a new object of class *AClass* for each object in *Students*. In the next query the result is obtained by creating new objects using the student objects and the sets of courses.

Q7. Return students and courses taken by them.

```
Set [ s <- Students | ACLASS.NEW(s, s.takes) ]
```

Nested Queries.

The result is obtained by creating new objects using the student objects and the sets of courses. Nested queries enable richer data structures to be returned and complex selection conditions to be expressed. An inner query above is a parameter to the method call in the result expression of the outer query.

Q8. Return students and courses taken by them with a credit rating over one.

```
Set [ s <- Students | ACLASS.NEW(s, Set[c<-s.takes;c.credits>1|c])]
```

Multiple Generators.

Multiple generators allow relationships that are not explicitly defined in the database schema to be reconstructed. Two variables can range over the same set independently.

Q9. Return students studying in the same department as SteveJ.

```
Set [ x <- Students; y <- Students; x.name == "SteveJ"; x.major == y.major | y]
```

Dependent Generators.

Dependent generator is used to facilitate querying over the elements in a nested collection.

Q10. Return courses taken by the students.

```
Set [ s <- Students; c <- s.takes | c ]
```

Literal Generators.

Collection literals can simplify queries by making them more concise and clearer.

Q11. Return those courses among C1,C2,C3 which have a credit rating over one.

```
Set [ c <- Courses; x <- Set ["C1","C2","C3"]; c.code == x ; c.credits > 1 | c ]
```

Existential Quantifiers.

A restricted form of existential quantification is provided by *some*, which can appear on either side of an operator. In Q12, the filter succeeds if course code is one of the memberslisted. In Q13, the filter returns true if there is a common element between the two sets; i.e. an non-empty intersection.

Q12. Return those courses among C1,C2,C3 which have a credit rating over one.

```
Set [ c <- Courses; c.credits > 1;
          c.code = SOME Set ["C1", "C2", "C3"] | c]
```

Q13. Return students taking a course given by Steve Johnson.

```
Set [ j <- StaffMembers; j.name = "SteveJohnson";
          s <- Students; SOME s.takes = SOME j.teaches | s]
```

Universal Quantifiers.

In the following query the keyword *EVERY* is the universal quantifier. The filter succeeds if all the course elements in *s.takes* are also in the set *j.teaches*; i.e. subset.

Q14. Return students taking only courses given by Steve Johnson.

```
Set [ j <- StaffMembers; j.name = "SteveJohnson";
          s <- Students; EVERY s.takes = SOME j.teaches | s]
```

Numerical Quantifiers.

In the following queries the keywords *ATLEAST*, *JUST* and *ATMOST* are the numerical quantifiers. Numerical quantifiers are very useful in dealing with duplicate elements in collections and the number of elements that are common between two collections, i.e. the size of intersection. In Q15, the filter turns true if there are two or more elements that are common between specified sets. In Q16, the filter succeeds if there are exactly two common elements. In Q17, the size of intersection must be less than or equal to two.

Q15. Return students taking two or more courses given by Steve Johnson.

```
Set [ j <- StaffMembers; j.name = "SteveJohnson";
          s <- Students; SOME s.takes = ATLEAST 2 j.teaches | s]
```

Q16. Return students taking exactly two courses given by Steve Johnson.

```
Set [ j <- StaffMembers; j.name = "SteveJohnson";
          s <- Students; SOME s.takes = JUST 2 j.teaches | s]
```

Q17. Return students taking no more than two courses given by Steve Johnson.

```
Set [ j <- StaffMembers; j.name = "SteveJohnson";
          s <- Students; SOME s.takes = ATMOST 2 j.teaches | s]
```

Aggregate Functions.

The aggregate function *size* returns the number of elements in a collection. It is defined for all collection classes. For bags and lists duplicate elements are included in the counting. Some aggregate functions are possibly defined only for certain collection classes.

Q18. Return courses with less than two assessments.

```
Set [ c <- Courses; c.assessments.size < 2 | c ]
```

Equality.

It is quite a necessity to have an ability to compare two collections based on the elements, occurences and their order. Thus two bags are equal if for each element drawn from either collection there is equal number of occurences in both bags. For the lists, number of occurences and the positions must be the same.

Q19. Return courses requiring no prerequisite courses.

```
Set [ c <- Courses ; c.prerequisites == Set [ ] | c ]
```

Occurrences and Counting.

Bags and lists allow duplicates. The following two queries are to show how the occurences of elements could be used.

Q20. Return courses with four 25% assessments.

```
Set [ c <- Courses; JUST 4 c.assessments = 25 | c ]
```

Q21. Return the number of assessments counted 25% in the course "db4".

```
Set [ i <- List{0..db4.assessments.size };
                              JUST i db4.assessments = 25 | i]
```

Positioning and Ordering.

A list allows duplicates and keeps track of the order of the elements. In Q22, the first two elements of the list are returned and used in a generator. In Q23, a sublist whose first element is Steve and whose last element is Bob is returned. It returns an empty list if Steve does not come before Bob in a supervisor list.

Q22. Return the first and the second supervisors of Steve Johnson.

```
Set [ s <- Students; s.name = "SteveJohnson";
     p <- s.supervisedBy.[1..2] | p ]
```

Q23. Return students having Steve before Bob in their supervisor lists.

```
Set [ s <- Students; s.supervisedBy.[Steve : Bob] ~== List [ ] | s]
```

Union.

The *union* operator combines two collections to form a new collection of the same class but having all the elements. The union of bags contains all the elements including the duplicates. The union of a list to another one appends the latter one to the former one.

Q24. Return students in the CS and EE departments.

```
Set [s <- Students; s.major.name = "CS" | s ]
    UNION Set [s <- Students; s.major.name = "EE" | s ]
```

Differ.

For the *differ*, the class of the result elements is determined in the same way as in union. The number of occurences for an element in the result collection is the difference of that in the operand collections. For lists, differ will remove the last match.

Q25. Return cities where students, but no staff, live.

```
Set [s <- Students | s.address.city ]
    DIFFER Set [s <- StaffMembers | s.address.city ]
```

Converting Collections.

There might be a need to convert a bag, a list or a set one to another. Bag to set conversion would eliminate duplicates while converting to a list would involve an additional effect of assigning an arbitrary order over the result elements.

Q26. Return the salary of tutors and keep the possible duplicate values.

```
Bag [ t <- Tutors | t.salary ]
```

Mixing Collections.

Object-oriented data model supports more than one kind of collection. Hence the corresponding query notation should support not only different collection classes but also the mix of them in the same query. In the following query, *s.supervisedBy* returns a list and is mixed with two generators drawing from sets. It should be mentioned that swapping of generators will not be allowable if the result collection is to be a list.

Q27. Return courses taught by the supervisors of Steve Johnson.

```
Set [ s <- Students; s.name = "SteveJohnson";
          sup <- s.supervisedBy; c <- sup.teaches | c]
```

Query Functions and Recursion.

It is natural to find the cyclic relationships in object-oriented data models. This implies recursion support. The recursive queries can be expressed in object comprehensions via query functions. In the following query the result is generated by retrieving elements from a collection returned by a recursive function. *f(c.prerequisites).* This function takes a set of courses and returns a set of courses. For each element drawn from the input collection. f is applied recursively on the prerequisite courses. and the result is then used as a part of the input. The recursion terminates when f is passed an empty set.

Q28. Return all direct and indirect prerequisite courses for the "DB4" course.

```
let f(cs : Set of Course ) be
        cs UNION Set [x <- cs; y <- f(x.prerequisites) | y]
    in Set[c <- Courses; c.code = "DB4"; p <- f(c.prerequisites) | p]
```

Chapter 3

Translation of OCL queries

This chapter provides manual translations of object comprehensions queries into O++ queries of ODE. The OCL sample queries are from chapter 2.

```
1. Set [ s <- StaffMembers ; s.salary > 1000 | s ]

Set<StaffMembers> tempSet;

for s in StaffMembers
    suchthat (s->salary > 1000)
    tempSet.add(s);


------------------------------------------------------
2. Set [ t <- Tutors ; t.address.city == "Glasgow" | t ]

Set<Tutors> tempSet;

for t in Tutors
    suchthat (t->address->city == String("Glasgow") )
    tempSet.add(t);


------------------------------------------------------
3. Set [ t <- Tutors; t.department = t.major | t ]
```

```
Set<Tutors> tempSet;

for t in Tutors
    suchthat (t->department == t->major)
    tempSet.add(t);

-----------------------------------------------------
4. Set [ p <- Persons ; p HASTYPE VisitingStaff | p ]

Set<Persons> tempSet;

for p in Persons
    suchthat (p is VisitingStaff *)
    tempSet.add(p);

-----------------------------------------------------------------------
5. Set [ p <- Persons ; p HASTYPE VisitingStaff WITH p.salary > 1000 | p ]

Set<Persons> tempSet;

for p in Persons
    suchthat ((p is VisitingStaff *) && (p.salary > 1000))
    tempSet.add(p);

--------------------------------------------------------------
6. Set [ s <- Students ; a AS s.major.address.street ; a = "Hill st"
        OR  a = "U ave" | s ]

Set<Students> tempSet;

for s in Students
    suchthat ((a = s->major->address->street) == String("Hill st") ||
                                       a == String("U ave") ) )
    tempSet.add(s);
```

```
--------------------------------------------------
7. Set [ s <- Students | ACLASS.NEW(s, s.takes) ]

Set <AClass> temp;

for s in Students
    temp.add( new AClass(s, s->takes));

---------------------------------------------------------------------
8. Set [ s <- Students | ACLASS.NEW(s, Set[c<-s.takes;c.credits>1|c])]

Set <AClass> temp1;

for s in Students
  {
    Set <Course> temp2;

    for c in s->takes
        suchthat ( c->credits > 1 )
        temp2.add(c);
    temp1.add( new AClass(s, temp2))
  }

------------------------------------------------------------------------------
9. Set [ x <- Students ; y <- Students ; x.name == "SteveJ" ; x.major == y.majo

Set<Students> tempSet;

for x in Students, y in Students
    suchthat (x->name == String("SteveJ") && x->major == y->major)
        tempSet.add(y);

------------------------------------------------------
```

```
10. Set [ s <- Students; c <- s.takes | c ]

Set<Courses> tempSet;

for s in Students
  for c in s->takes
    tempSet.add(c);

-------------------------------------------------------
11. Set [ c <- Courses ;
          x <- Set [ "C1" , "C2" , "C3" ] ;
          c.code == x ; c.credits > 1 | c ]


Set<String>  tempSet;

for c in Courses
{
      Set<Courses> tempSet1;

      tempSet1.add("C1");
      tempSet1.add("C2");
      tempSet1.add("C3") ;

      for x in tempSet1
           suchthat (c->code == x && c->credits > 1 )
           tempSet.add(c);
}

-------------------------------------------------------
12. Set [ c <- Courses ; c.credits > 1 ;
          c.code == SOME Set [ "C1" , "C2" , "C3" ] | c ]
```

```
Set<String>  tempSet;

for c in Courses
   suchthat (c.credits > 1)
  {

   Set<Courses> tempSet1;

   tempSet1.add("C1");
   tempSet1.add("C2");
   tempSet1.add("C3");

   if (tempSet1.member(c.code))
      tempSet.add(c);

  };
```

```
------------------------------------------------------
13. Set [ j <- StaffMembers ; j.name = "SteveJohnson" ;
          s <- Students ; SOME s.takes = SOME j.teaches | s ]


Set<Students> tempSet;

for j in StaffMembers
      suchthat(j.name == String("SteveJohnson"))
      for s in Students
            suchthat(s.takes.intersection(j.teaches))
            tempSet.add(s);

------------------------------------------------------
14. Set [ j <- StaffMembers ; j.name == "SteveJohnson" ;
          s <- Students ; EVERY s.takes == SOME j.teaches | s ]
```

```
Set<Courses> tempSet;

for j in StaffMembers
      suchthat(j.name == String("SteveJohnson"))

      for s in Students
            suchthat(s.takes.every(j.teaches))
            tempSet.add(s);

-----------------------------------------------------
15. Set [ j <- StaffMembers ; j.name == "SteveJohnson" ;
          s <- Students ; SOME s.takes == ATLEAST 2 j.teaches | s ]


Set<Courses> tempSet;

for j in StaffMembers
      suchthat(j.name == String("SteveJohnson"))

      for s in Students
            suchthat(j.teaches.atleast(2, s.takes))
            tempSet.add(s);

-----------------------------------------------------
16. Set [ j <- StaffMembers ; j.name == "SteveJohnson" ;
          s <- Student s; SOME s.takes == JUST 2 j.teaches | s ]


Set<Courses> tempSet;

for j in StaffMembers
      suchthat(j.name == String("SteveJohnson"))

      for s in Students
```

```
            suchthat(j.teaches.just(2, s.takes))
            tempSet.add(s);


-------------------------------------------------------
17. Set [ j <- StaffMembers ; j.name == "SteveJohnson" ;
          s <- Students ; SOME s.takes == ATMOST 2 j.teaches | s ]


Set<Courses> tempSet;

for j in StaffMembers
      suchthat(j.name == String("SteveJohnson"))

      for s in Students
            suchthat(j.teaches.atmost(2, s.takes))
            tempSet.add(s);


-------------------------------------------------------
18. Set [ c <- Courses ; c.assessments.size < 2 | c ]


Set<Courses> tempSet;

for (c in Courses)
      suchthat (c.assessments.size < 2)
      tempSet.add(c);


-------------------------------------------------------
19. Set [ c <- Courses ; c.prerequisites == Set [ ] | c ]


Set<Courses> tempSet;

for c in Courses
```

```
{
    Set<Courses> tempSet1;

    if (c.prerequisites == tempSet1)
    tempSet.add(c);
}

-----------------------------------------------------
20. Set [ c <- Courses ; JUST 4 c.assessments == 25 | c ]

Set<Courses> tempSet;

for c in Courses
    suchthat(c.assessments.frequency(25) == 4)
    tempSet.add(c);

-----------------------------------------------------
21. Set [ i <- List{0..db4.assessments.size };
                      JUST i db4.assessments = 25 | i]

Set<int> tempSet;

for c in Courses
     suchthat (c->code == String("db4"))
     {
       List<Int> tempList;
       for i in tempList.intlist(0, c->assessments.size);
             suchthat(frequency(c->assessments, 25) == i)
             tempSet.add(i);
     }

-----------------------------------------------------
22. Set [ s <- Students; s.name = "Steve Johnson";
          p <- s.supervisedBy.[1..2] | p ]
```

```
Set<StaffMembers> tempSet;

for s in Students
    suchthat (s->name == String("Steve Johnson"))
    {
      List<StaffMembers> tempList;

      tempList = s->supervisedBy.sublist(1, 2);
      for p in tempList
          tempSet.add(p);
    };

------------------------------------------------------
23. Set [ s <- Students ;
          s.supervisedBy.[ Steve : Bob ] ~= List [ ] | s ]


Set<Students> tempSet;

for s in Students
{
   List<StaffMembers> tempList;
   if (s.supervisedBy.ssublist("Steve", "Bob") != tempList)
   tempSet.add(s);
};

------------------------------------------------------
24. Set [s <- Students ; s.major.name = "CS" | s ]
    UNION Set [s <- Students ; s.major.name = "EE" | s ]


Set<Students> tempSet1;
```

```
Set<Students> tempSet2;

for s in Students
    suchthat (s.major.name == String("CS"))
    tempSet1.add(s);

for s in Students
    suchthat (s.major.name == String("EE"))
    tempSet2.add(s);

tempSet1.union(tempSet2);

------------------------------------------------------
25. Set [ s <- Students | s.address.city ]
    DIFFER Set [ s <- StaffMembers | s.address.city ]


Set<String> tempSet1;
Set<String> tempSet2;

for s in Students
    tempSet1.add(s.address.city);
for s in StaffMembers
    tempSet2.add(s.address.city);

tempSet1.differ(tempSet2);

------------------------------------------------------
26. Bag [ t <- Tutors | t.salary ]


Bag<Int> tempBag;

for t in Tutors
```

```
    tempBag.add(t.salary);

------------------------------------------------------
27. Set [ s <- Students ; s.name == "SteveJohnson" ;
          sup <- s.supervisedBy ; c <- sup.teaches | c ]


Set<Courses> tempSet;

for (s in Students)
    suchthat(s.name == "Steve Johnson"))
    for (sup in s.supervisedby)
        for (c in sup.teaches)
              tempSet.add(c);


------------------------------------------------------
28. let f(cs : Set of Course ) be
        cs UNION Set [x <- cs; y <- f(x.prerequisites) | y]
    in Set[c <- Courses; c.code = "DB4"; p <- f(c.prerequisites) | p]


// prototype of f
   Set<Courses> f (Set<Courses> cs);

Set<Courses> tempSet;

for c in course
    suchthat (c->code == String("DB4"))
    for p in f(c->prerequisites)
          tempSet.add(p);


//  definition of f
    Set<Courses> f (Set<Courses> cs)
```

```
{
 Set<Courses> tempS;

 for x in cs,
       for y in f(x->prerequisites)
             tempS.add(y);
 return cs.union(tempS);
}
```

Chapter 4

Translator

This chapter elaborates the issues of translation from OCL to O++ language. It discusses the implementation of the translator, the actual code of which is provided in appendix A (*.h files) and appendix B (*.C files).

The issues of translator and compiler design are well addressed in [1] and [3].

A program that takes as an input a program written in one language and produces as an output a program in another language is called *translator* for obvious reasons. The language of an input program is then called *source language*. The language of an output program is called *target* or *object language*. Translators which convert a high-level language to machine code called *compilers*. Systems which are virtually the same as compilers, except that instead of generating machine code for later execution, they transfer control to a routine for actual execution upon recognition of each statement called *interpreters*. In our case OCL is the source language and O++ is the target. Thus we translate one high-level language to another.

4.1 The phases of translation

The translation is realized by *translate* method of an object of *class Manager*. Due to its complexity, it is reasonable from an implementational point of view as well as logical viewpoint to divide the process of translation into a number of phases. The output of each phase is passed as an input to the next phase.

4.1.1 Lexical Analyzer

Lexical Analyzer, sometimes called *Scanner*, is the first phase. It examines the characters of the source language and groups them into units they logically belong together. Such units are called *tokens*. They are defined in *token.h* file. To handle token objects *class token* is introduced in this file. The implementation of the lexical analyzer is realized by the *class lex*. The functionality of an object of the class lex is to *getNextToken*, *putbackToken* and *printToken* as it could be seen from *lex.h*. The implementation of these methods is in *lex.C*. The method *printToken* is there only for verification purposes. For the sake of demonstrating different token types and verifying lexical analyzer we present here the following example.

Example 1

Input:

```
Set [ s <- Students ; a AS s.major.address.street ;
    a == "Hillst" ; a == "Uave" | s ]
```

Output:

```
token   < Set >
token   < lftsqbracket >
token   < identifier  s >
token   < arrow >
token   < identifier  Students >
token   < semicolon >
token   < identifier  a >
token   < as >
token   < identifier  s >
token   < dot >
token   < identifier  major >
token   < dot >
token   < identifier  address >
token   < dot >
token   < identifier  street >
token   < semicolon >
token   < identifier  a >
token   < equal >
token   < identifier  "Hillst" >
token   < semicolon >
token   < identifier  a >
token   < equal >
token   < identifier  "Uave" >
token   < bar >
token   < identifier  s >
token   < rgtsqbracket >
token   < endtoken >
```

4.1.2 Parser

Thus the output of the lexical analyzer is a stream of tokens, which is an input for the next phase, the *syntax analyzer* or *parser.* Parsing is accomplished by the *parse* method of *Manager* object. The top-down parsing technique, so called recursive descent parsing was used for that purpose. A parser groups tokens together into syntactic structures. As it could be seen in the Grammar (Chapter 5) *qualifier* is the main such structure in our case. There are three types of a qualifier: generator, localdef and filter. The job of recognizing and differentiating a qualifier is again on an object of the *class Manager*, which is implemented in *Manager.h* and *Manager.C* files. The method to do the job is *recognizeGLF.* It should be mentioned that Manager object is a mainspring that controls and coordinates the mutual performance of all other objects. The output of a parser is a so called *Parse Tree*, a syntactic structure that incorporates all recognized structures. In our case the main part of a tree is created by Manager's method *implant.* Tree has Collection type, Result node and a list of nodes, so called generatorNodes. Each of them contains a generator and two (possibly empty) children lists: a list of localdefNodes and a list of filterNodes. These nodes contain localdefs and filters respectively. The hierarchy of nodes is shown on the figure 3. All these nodes are implemented in *nodes.h* file.

Let us continue the previous example, assuming that the stream of tokens was fed

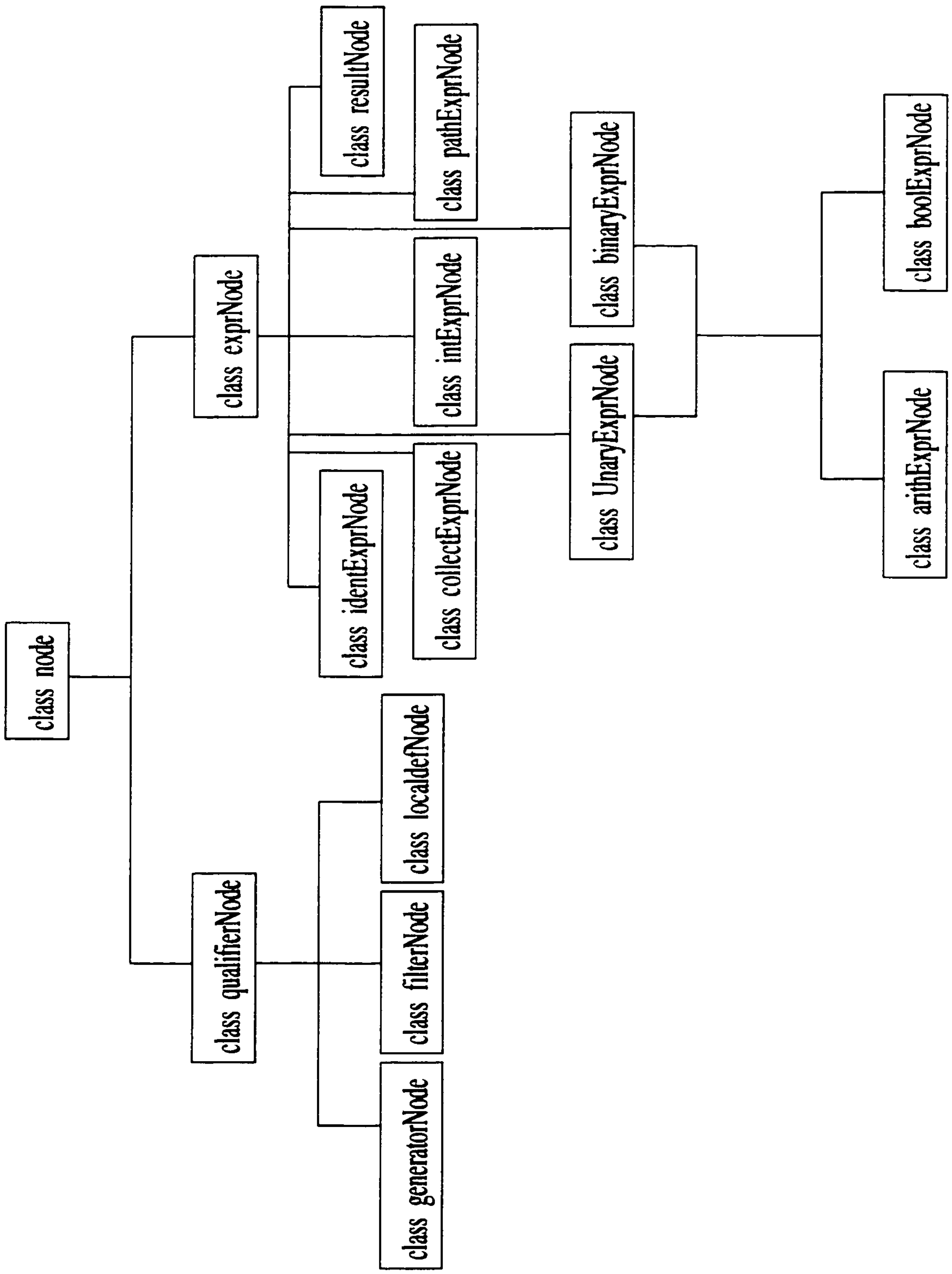

Figure 3: Nodes Hierarchy.

into the parser. Thus the input is the output shown in the Example 1. The figure 4 presents the output parse tree.

4.1.3 Intermediate code generation and optimization

The next phase is the *intermediate code generation* which produces intermediate code from the parse tree. Intermediate code is then fed into *Code Optimizer* that is the engine of the next phase. namely *code optimization.* This phase is to improve the intermediate code so that the ultimate resulting code is to be more efficient. faster in run and possibly to take less space. These two phases are skipped in our translator. Code optimization is left for future work.

4.1.4 Code generation

Code generation is the final phase. Designing an efficient code generator as it is noted in [1] is one of the most difficult parts both practically and theoretically. This is obvious for compilers. a particular case of translators when the source is a high-level language (e.g. PL1 etc.) and the target is a low-level language such as an assembly or machine languages. In our case it is a matter of printing O++ statements. The Manager's method *codegen* is to fulfill this task. As you might see in Chapter 4 it could be divided into three subtasks: printing declarations. printing *for* loop with *suchthat* statement and printing result. That is why *codegen* calls *buildDECLARATION*, *buildFORSUCHTHAT* and *buildRESULT* methods of Manager. The first one prints declaration statement checking the collection type and result node. The second one prints *for* loop for each visited non-empty generatorNode in the list of qualifiers. followed by *suchthat* statement which is based on the information from tracing a local list of localdefs and a local list of filters. *buildRESULT* prints the result of a query. Let's resume our example. Assuming the parse tree on Figure 4 to be the input, the output would be as follows.

Example 3.

Output:

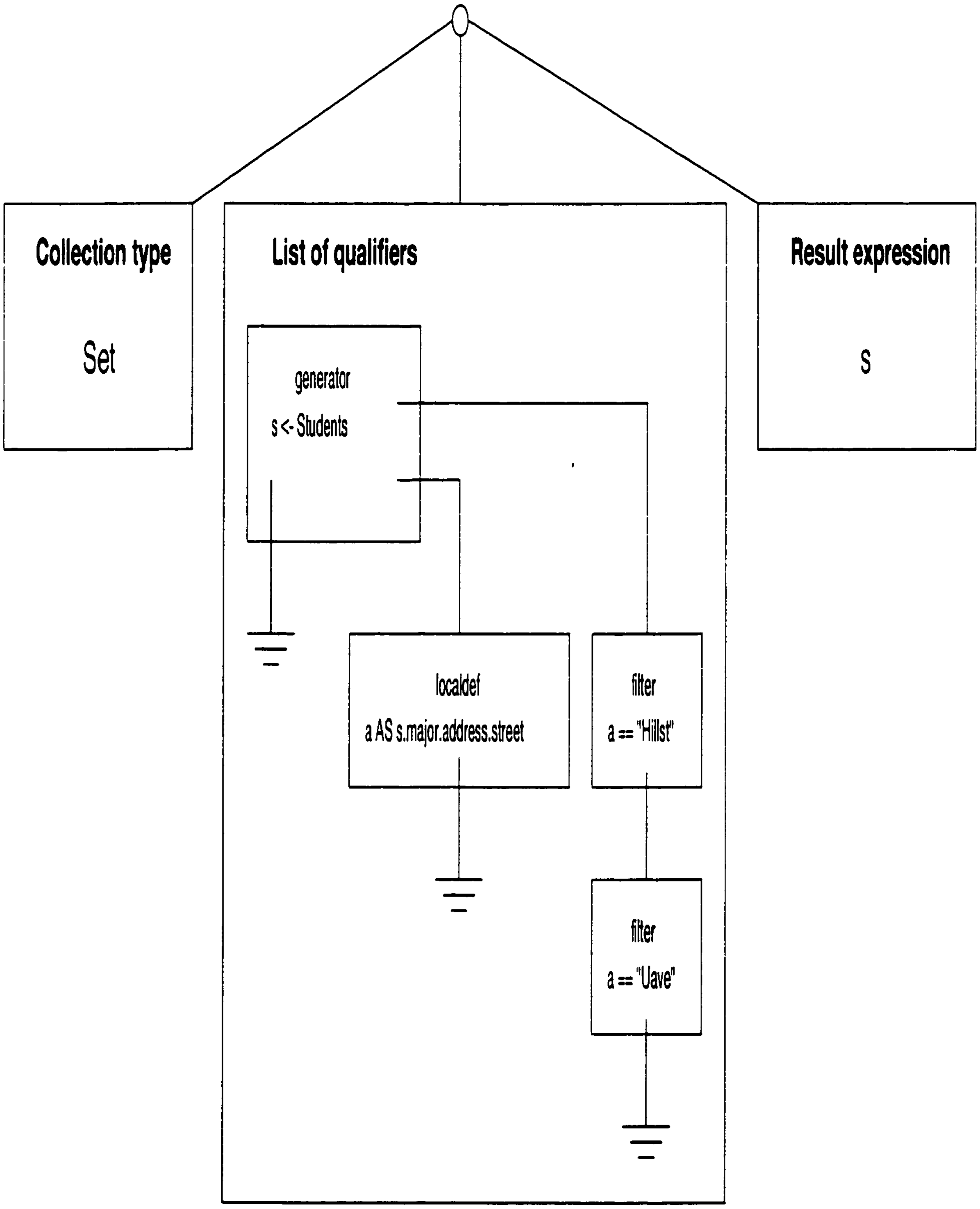

Figure 4: Example 2: The Parse Tree.

```
Set<Students> templ;
for (s in Students)
    suchthat ( ((a = s.major.address.street) == "Hillst") && (a == "Uave") )
        templ.add(s);
```

This is the desired O++ query translation. The translating process described above is summarized in the Figure 5.

4.1.5 Table-management and Error handling

It should be mentioned that all phases are involved in *bookkeeping* or *table-management* activity. This is to keep track of the names used by the program and to record their type information. *Symbol Table* is used to keep this information. Thus the information about data is collected by lexical and syntactical analyzers and entered into the symbol table. An object of *class hashTable* handles the functionality of a symbol table. Although symbol table can be organized in many ways, as the name suggests *hash coding* was utilized since it is generally most preferred and accepted method of handling symbol tables as [1] states due to the efficiency reasons. Hash coding, in short, uses some computable function of the numeric representation of the name to determine at which point in the list the name should be entered. This is accomplished by the method *hash* of class *hashTable*. Files *table.h* and *table.C* contain the code. *Error Handling* is another activity of all phases. The error handler is to be invoked whenever a flaw in the source is detected. Then a warning error message is to be issued. There is no error handler in our translator at this time. Error handling is done by printing a message by any routine that discovers a flaw. Again, both table-management and error handling interact with all phases.

4.1.6 Passes

There are *multi-pass* and *single-pass* translators. A pass reads the source, which can be the output of the previous pass, makes the transformations specified by its phases and writes the output to an intermediary file. This file is to be read by the next pass. Obviously that a multi-pass translator is slower than a single-pass translator. Our translator is a single-pass translator.

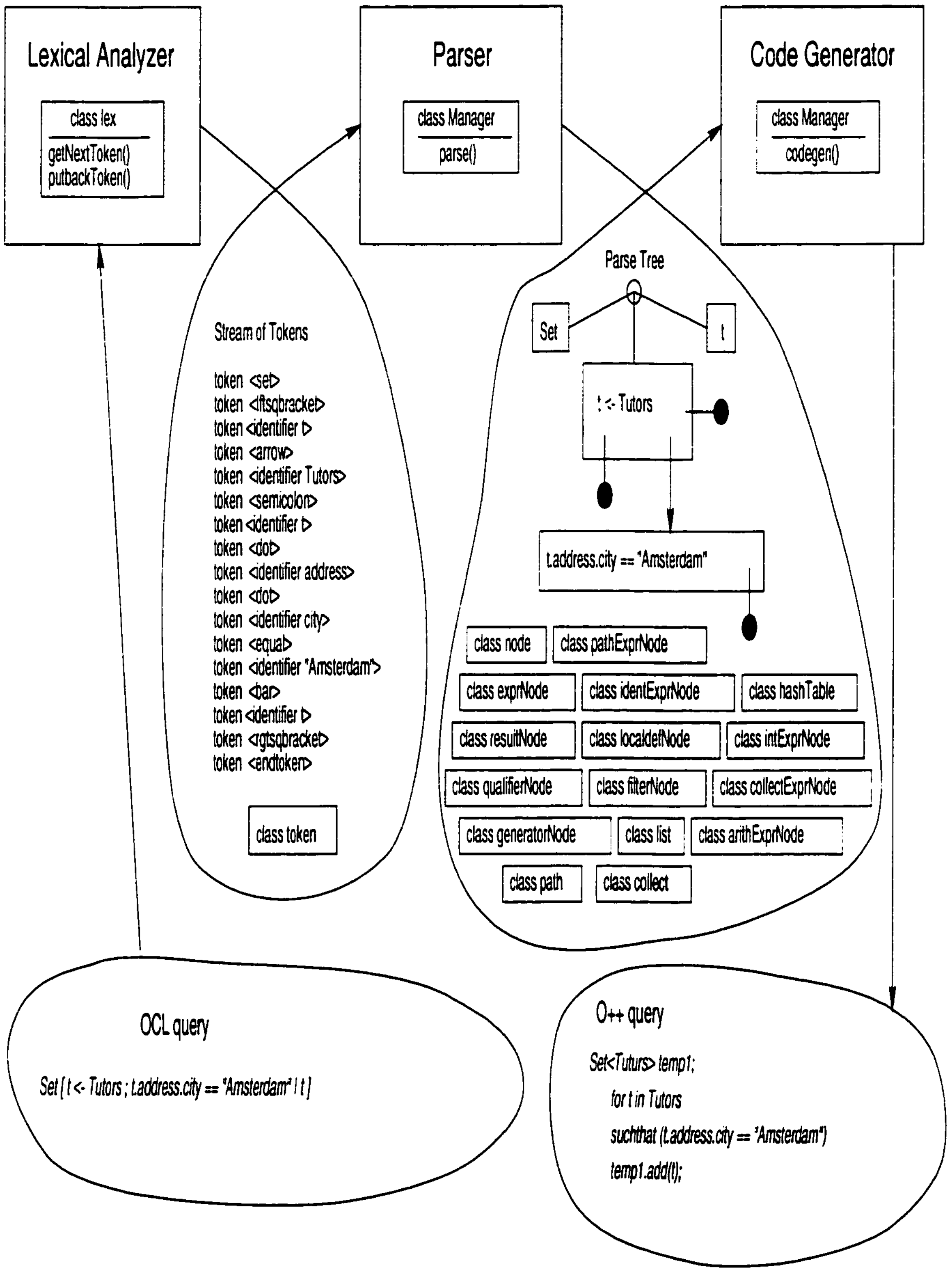

Figure 5: Translator Architecture.

4.2 Class Dictionary

This translator is coded in C++ language and hence everything is done by objects of different classes. I already mentioned some classes in this chapter while explaining the translation phases. The hierarchy of nodes classes is displayed in figure 3 and all main classes are also presented in figure 5. This section is to exhibit all created classes. In the following dictionary a class name is followed by the description of a class. Figure 6 compliments the dictionary with a brief overview of all classes.

- **class arithExprNode**
  Defined in *nodes.h* file. Inherits from both *binaryExprNode* and *unaryExprNode* classes. Used to represent arithmetical expressions.

- **class binaryExprNode**
  Defined in *nodes.h* file. Inherits from *exprNode* class. Used as a holder of binary expressions. Base class for *arithExprNode* and *boolExprNode* classes.

- **class boolExprNode**
  Defined in *nodes.h* file. Inherits from both *binaryExprNode* and *unaryExprNode* classes. Used to represent boolean expressions.

- **class collect**
  Defined in *collect.h* and *collect.C* files. Used to represent "constant" collections, e.g. `Set [ t, p, k ]`. An object of *class collect* is utilized in *class collectExprNode*.

- **class collectExprNode**
  Defined in *nodes.h* file. Inherits from *exprNode* class. Used as a holder of "collect" expressions. As a particular case of *exprNode* utilized in objects of class *qualifierNode* and its subclasses, namely *generatorNode*, *localdefNode* and *filterNode*.

- **class exprNode**
  Defined in *nodes.h* file. Inherits from *node* class. Base class for the following classes: *identExprNode*, *intExprNode*, *pathExprNode*, *resultNode*, *collectExprNode*, *binaryExprNode* and *unaryExprNode*. Utilized in objects of class *qualifierNode* and its subclasses, namely *generatorNode*, *localdefNode* and *filterNode*.

Figure 3.

- **class filterNode**

  Defined in *nodes.h* file. Inherits from *qualifierNode* class. Used as a holder of a filter, that is a particular form of a qualifier.

- **class generatorNode**

  Defined in *nodes.h* file. Inherits from *qualifierNode* class. Used as a holder of a generator, that is a particular form of a qualifier. In addition an object of *generatorNode* class has two lists of *qualifierNode* objects: localdefList and filterList. One of the main components of Parse Tree.

- **class hashTable**

  Defined in *table.h* and *table.C* files. Used to represent Symbol Table. Functionality: insert an object of class *record* into Symbol Table; look for info in Symbol Table. A "friend" of *record* class.

- **class identExprNode**

  Defined in *nodes.h* file. Inherits from *exprNode* class. Used as a holder of identifier expressions. As a particular case of *exprNode* utilized in objects of class *qualifierNode* and its subclasses, namely *generatorNode*, *localdefNode* and *filterNode*.

- **class intExprNode**

  Defined in *nodes.h* file. Inherits from *exprNode* class. Used as a holder of integer expressions. As a particular case of *exprNode* utilized in objects of class *qualifierNode* and its subclasses, namely *generatorNode*, *localdefNode* and *filterNode*.

- **class lex**

  Defined in *lex.h* and *lex.C* files. Used to accomplish Lexical Analyzer. Functionality: produce token stream by using *getNextToken* and *putbackToken* methods. Uses objects of class *token*.

- **class List**

  Defined in *list.h* file. Template class. Used to provide a list facilities for *generatorNode* and *Manager* classes. Uses objects of class *listNode*. A "friend" of *listNode* class.

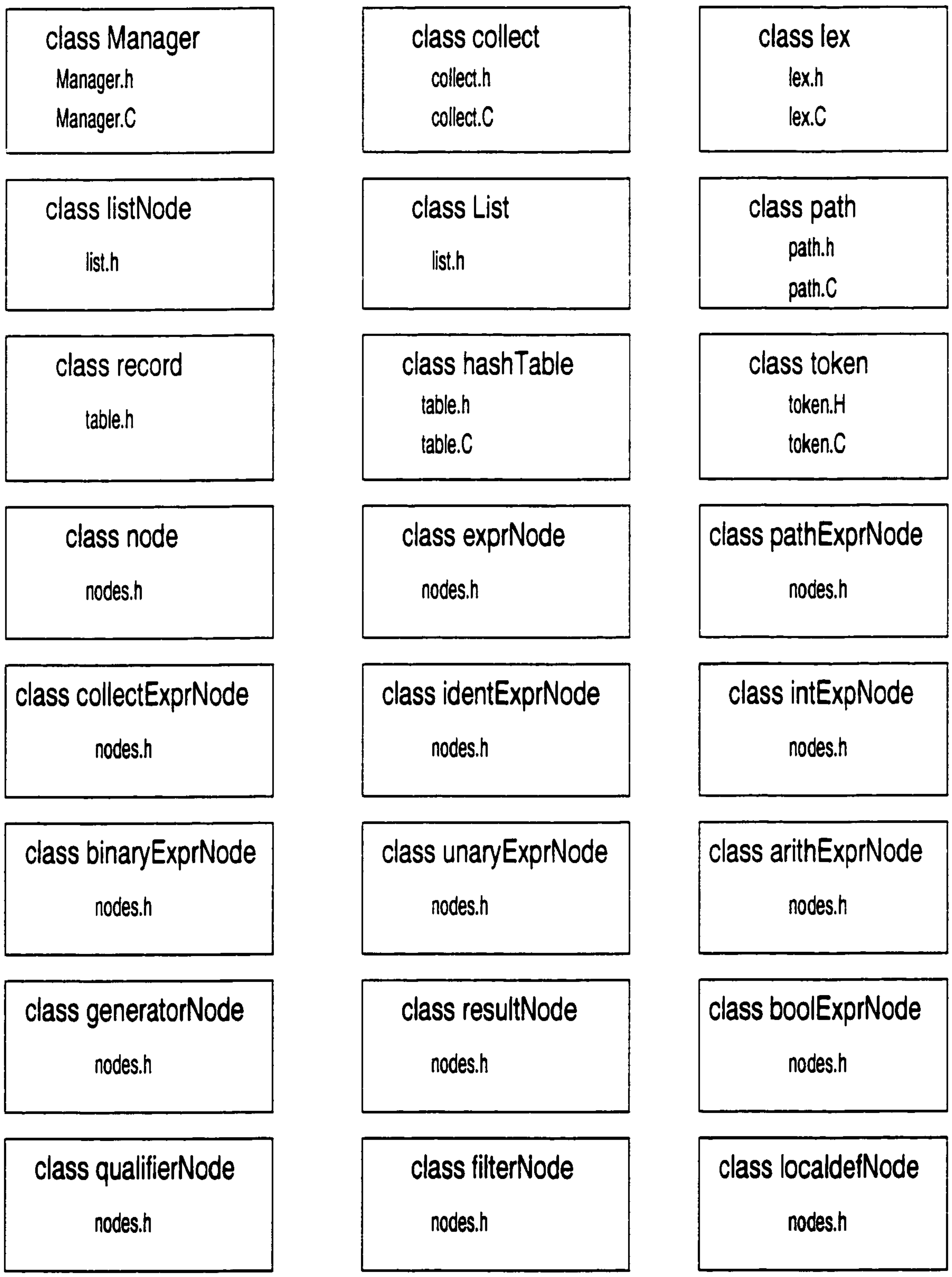

Figure 6: Overview of classes.

- **class listNode**

  Defined in *list.h* file. Template class. Used by objects of class *List*. Declares class *List* to be its "friend".

- **class localdefNode**

  Defined in *nodes.h* file. Inherits from *qualifierNode* class. Used as a holder of a localdef, that is a particular form of a qualifier.

- **class Manager**

  Defined in *Manager.h* and *Manager.C* files. An object of class Manager is a mainspring that controls and coordinates the mutual performance of all objects of all other classes in order to provide translation. For that purpose it has *translate* method which calls *parse* and *codegen* methods to accomplish parsing and code generation phases. In its turn *parse* calls for the *implant* method and a number of *recognize-* methods such as *recognizeCollectionType*, *recognizeGLF* and *recognizeResult*. In their turn they use *recognizeIdentifier*, *recognizePathname* and *recognizeQuantifier* methods. The *codegen* method invokes *build-* methods: *buildDECLARATION*, *buildFORSUCHTHAT* and *buildRESULT*. The *builds* run *print-* methods: *printCollectionType*, *printResult*, *printPath*, *printNode*, *printGen*, *printFil*, *printFilter* and *printLocaldef*. There is *thereAfter* method which is used by "main" program in *main.C*. This method is to verify whether there exists "union" or "differ" conjunction that conjoins another query to the current one. A "friend" of *qualifierNode* class.

- **class node**

  Defined in *nodes.h* file. An abstract class. Base class for *exprNode* and *qualifierNode*. Figure 3.

- **class path**

  Defined in *path.h* and *path.C* files. Used to represent pathname, e.g. `t.address.city`. An object of *path* class is utilized by objects of class *pathExprNode*.

- **class pathExprNode**

  Defined in *nodes.h* file. Inherits from *exprNode* class. Used as a holder of "path" expressions. As a particular case of *exprNode* utilized in objects of class *qualifierNode* and its subclasses, namely *generatorNode*, *localdefNode* and *filterNode*.

- **class qualifierNode**
  Defined in *nodes.h* file. Inherits from *node* class. Base class for *localdefNode*, *generatorNode* and *filterNode* classes. Declares class *Manager* to be its "friend".

- **class record**
  Defined in *table.h* file. Utilized by objects of class *hashTable*. Used to represent Symbol Table entry. Declares class *hashTable* to be its "friend".

- **class resultNode**
  Defined in *nodes.h* file. Inherits from *exprNode* class. Used as a holder of result expression.

- **class token**
  Defined in *token.h* and *token.C* files. Used to represent tokens.

- **class unaryExprNode**
  Defined in *nodes.h* file. Inherits from *exprNode* class. Used as a holder of unary expressions. Base class for *arithExprNode* and *boolExprNode* classes.

Chapter 5

Conclusion

The original idea was to study and implement OCL. Object Comprehension language. the new powerful query notation.

This idea initialized a team work which consisted of three interrelated projects. The main goal of these three projects was to test OCL so a user would be able to run OCL queries against some certain database and view the results. First of all the required database system is to be built and managed. This constitutes one of components. the database project. Building a database system in turn requires a symbol table for such a database, which describes a plot for another component, the schema project. This project is responsible for the implementation of the above mentioned symbol table. The third team work component was to provide the translation of user specified OCL queries into an interface query language that could be recognized by and run against the target database created in the database project. Based on this landscape we can have a closer look at each of these components and their mutual relationships.

The essence of the database project was to set up and manage the University sample database. It was decided that the target database is to be based on Ode, experimental object-oriented database system from AT&T. It is defined, queried and manipulated in O++. the database interface programming language which is based on C++ programming language. The University sample database reflects a university model, a schema of interrelated objects of O++ classes such as Department, Staff, Student, Course, etc. The O++ based queries are to be run against the University sample database. The results are obtained by utilizing the Ode built-in query facilities and implemented O++ classes to support OCL concepts, so called data structure utilities for query processing such as Set, List and Bag. The query results then would be printed in the form of collection of solutions.

The purpose of the schema project was to implement a symbol table for the above mentioned University sample database of the database project. It was to look into BNF grammar for the schema definitions and implement the facility to input views described as schema's for those OCL queries which create objects of new classes .

This project is focusing on the translator implementation and translation of the OCL queries into O++, an interface language for the Ode database system. Those OCL-to-O++ translated queries were to be run against the University sample database built in the database project. The scope of the translation project was set to design and implement experimental OCL translator. The issues of translator's code optimization, error handling, user interface as well as the topic of Object Comprehension queries optimization were not in the scope of this project. The following project goals were achieved:

1. the background and the reason d'etre of OCL are presented; the grammar for OCL is written in BNF form.

2. The Ode concepts of O++ objects, transactions in O++ and events and triggers are outlined. Based on the Ode/O++ overview, the sample data model is presented.

3. Based on the sample data model, the sample queries are presented to display the collection support and object-orientation of OCL, structuring and computational power of it.

4. for each of the OCL queries the O++ translations are created. These manual translation versions, suitable for discussion, were needed to validate the understanding of the abilities of OCL as the query notation, the flexibility of the O++ interface language and the potential connectivity to the Ode-based University sample database.

5. The experimental OCL-to-O++ translator was implemented. The appendices contain the actual code. It is demonstrated that the output of the translator matches with the O++ queries that were manually translated. The implemented translator translates the queries with:

- Support of Object-Orientation (Q1 - Q6).
- Multiple (Q9), dependent (Q10) and literal (Q11) generators.
- Existential (Q12, Q13), universal (Q14) and numerical (Q15, Q16 and Q17) quantifiers.
- Support of Collection (Q18 - Q20, Q23 - Q27).
- Method calling (Q1) and its extension such as path expressions (Q2).
- *HASTYPE* class hierarchy (Q4).
- Object identity (Q3).
- Local definitions (Q6).
- Aggregate functions (Q18).
- Duplicate occurences in bags and lists (Q20).
- Elements ordering in lists (Q23).
- *UNION* (Q24) and *DIFFER* (Q25) operators.
- Converting (Q26) and mixing (Q27) collections.

Out of twenty eight originally suggested OCL queries twenty four are adequately translated.

The current translator version does not handles certain types of queries. This is left for future work. These are the following queries:

- Queries that require the creation of the new objects specified in a result expression (Q7).
- Nested queries (Q8).
- Queries with list elements positioning (Q22).
- Recursive queries (Q28).

As it can be seen all three projects were highly dependent upon each other. The translation component explicitly depends upon the database component since this is the final way to test the quality and quantity of provided OCL-to-O++ translation as to run translated queries against the target database. The database project relies on the symbol table of schema project. The outcome of it is also vital for the translator in regard to the queries which create new classes' objects. Obviously the aspect of projects' interdependency imposed certain challenge on proper organization of communication and cooperation between team members which influenced time frame of all three projects accordingly. This experience taught us the invaluable lesson that the success of a team project is a function of the proper and timely coordination between team members. The task of connecting and organizing components in one frame is left for the future work.

Eventually it will be desirable to embed OCL queries in C++ programs as a way to create applications, and to have a GUI interface for browsing databases using OCL. Before that the Unix shell scripts are the possible solution to connect the OCL-to-O++ translator and the Ode-based University sample database.

Bibliography

[1] Alfred V. Aho and Jeffrey D. Ullman. Principles of compiler design. Addison-Wesley, 1977.

[2] F. Bancilhon. Query languages for object-oriented database systems: Analysis and proposal. In *Proceedings of the GI Conference on Database Systems for Office, Engineering, and Scientific Applications*, pages 1–18, Springler-Verlag, 1989.

[3] Lynn Robert Carter and William M. Waite. An introduction to compiler construction. Harper-Collins, 1993.

[4] Daniel K. C. Chan and Philip W. Trinder. Object comprehensions: A query notation for object-oriented databases. In *Proceedings of the 12th British National Conference on Databases*, Guildford, Springler-Verlag, July 1994.

[5] Servio Logic Development Corporation. Programming in opal. version 1.3. 1987.

[6] D. J. Harper, D. K. C. Chan, and P. W. Trinder. A case study of object-oriented query languages. In *Proceedings of the International Conference on Information Systems and Management of Data*, pages 63–86, Indian National Scientific Documentation Centre (INSDOC), 1993.

[7] G. Pelagatti, E. Bertino, M. Nagri, and L. Sanella. Object-oriented query languages: The notion and the issues. In *IEEE Transactions on Knowledge and Data Engineering*, pages 223–237, June 1992.

[8] P. Hudak and P. Walder. Report on the functional programming language haskell. In *Technical Report 89/R5*, University of Glasgow, U.K., 1990.

[9] Ontologic Inc. Ontos sql guide. USA, 1991.

[10] S. Peyton-Jones. The implementation of functional programming languages. In *Chapter 7*, pages 127–138, Prentice-Hall, 1987.

[11] R.M. Burstall, D.B. MacQueen, and D.T. Sanella. Hope: an experimental applicative language. In *In Proceedings of the 1st ACM Lisp Conference*, pages 136–143, ACM Press, 1980.

[12] D. A. Turner. Recursion equation as a programming language. In *Functional Programming and its Application*, Cambridge University Press, 1981.

[13] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proceedings of the 2nd Conference on Functional Programming Languages and Computer Architectures*, pages 1–16, Springer-Verlag, 1985.

Appendix A

Header files: *.h

```
// token.h : token class definition

#ifndef TOKEN
#define TOKEN

#define MAXSTR 42  //number of elements in string[] and in tokenKind[]
#define FILMAX 6   //from 0 to FILMAX are the valid Filter operators

enum tokenType { lessthan, greaterthan, equal, lessORequal, greaterORequal,
                 notequal,  hastype, integer, identifier, arrow,
                 set, bag, list, semicolon, bar, comma, lftsqbracket,
                 rgtsqbracket, lftcurbracket, rgtcurbracket, dot,
                 lftpar, rgtpar, minus, colon, plus, multiply,
                 divide, endtoken, with, and, or, as, some,
                 every, just, atleast, atmost, reunion, differ, not };

typedef struct {tokenType quantifierType; int quantifierNumber;} quantif;

static char* string[] = {"Set", "Bag", "List", "[", "]", "{", "}",
                  "(", ")", "-", "+", ":", ";",
                  "|", ",", ".", "<-", "==", "<", ">", "<=", "=<", ">=",
                  "=>", "~=", "!=", "<>", "><", "*", "/", "HASTYPE",
                  "SOME", "EVERY", "JUST", "ATLEAST", "ATMOST",
                  "WITH", "AND", "OR", "AS", "UNION", "DIFFER", "~", "!", "NOT"

static tokenType tokenKind[] = {set, bag, list, lftsqbracket,
                         rgtsqbracket, lftcurbracket,
                         rgtcurbracket, lftpar, rgtpar, minus, plus,
                         colon, semicolon, bar, comma, dot, arrow, equal,
                         lessthan, greaterthan, lessORequal, lessORequal,
                         greaterORequal, greaterORequal, notequal,
                         notequal, notequal, notequal, multiply, divide,
                         hastype, some, every, just, atleast, atmost,
                         with, and, or, as, reunion, differ, not, not, not};

class token {
       tokenType type;
       char*     svalue;
       int       ivalue;

  public:
      token() {}
      token(tokenType t)          :type(t), svalue(0), ivalue(0) {}
      token(tokenType t, int i)   :type(t), svalue(0), ivalue(i) {}
      token(tokenType t, char* s) :type(t), svalue(s), ivalue(0) {}
      ~token() {}
      tokenType getTokenType() {return type;}
      int getIvalue()          {return ivalue;}
      char* getSvalue()        {return svalue;}
      friend ostream& operator << (ostream&, const token&);
};

#endif
/*************************************************************/
// path.h :  path variable definition (ex.:  time.date.month.year)

#ifndef PATH
```

```
#define PATH

#include <iostream.h>

class path {
   char* identValue;
   path* next;

 public:
   path()                :identValue(0), next(0) {}
   path(char* i)         :identValue(i), next(0) {}
   ~path()               {}
   void putIdentValue(char* i){identValue = i;}
   void putNext(path* n) {next = n;}
   path* getNext()       {return next;}
   char* getIdentValue() {return identValue;}
   char* getPathType();
   friend ostream& operator << (ostream&, const path&);
};

#endif
/************************************************************/
//list.h: listnode and list definitions

#ifndef LIST
#define LIST

#include <assert.h>

template < class T >
class listNode {
   friend class List<T>;
      T *container;
      listNode<T> *next;
   public:
      listNode(T *q) :container(q), next(0) {}
      ~listNode() {}
      void putNext(listNode<T>* n) {next = n;}
      listNode<T>* getNext() {return next;}
      T* getContainer() {return container;}
      T  getBucket()    {return *container;}
};

template < class T >
class List {
      listNode<T>* head;
      listNode<T>* last;

   public:
      List() :head(0), last(0) {}
      List(listNode<T>* n) :head(n), last(n) {}
      ~List() {head = 0; last = 0;}
      int empty();
      void append(T*);
      void display();
      void remove(T*);   //remove all occurencies
      listNode<T>* getHead() {return head;}
      void putHead(listNode<T>* h) {head = h;}
      void putLast(listNode<T>* l) {last = l;}
      void updateLastNext(listNode<T>* ln) {last->next = ln;}
      listNode<T>* getLast() {return last;}
};

template < class T >
int List<T>::empty()
{
  if (head) return(0);
  return(1);
}

template < class T >
void List<T>::append(T *q)  //put at the end of the list
{
  listNode<T> *p = new listNode<T>(q);

  assert(p != 0);

  if (head == 0) {head = p;}
  else {last->next = p;}
  last = p;
}

template < class T >
void List<T>::display()
{
  listNode<T> *p;
  for ( p = head; p; p = p->next)
     {
        cout << p->container->getType() << ": " << "( ";
        (p->container->getFirst())->print();
        cout << " " << p->container->getOp() << " ";
        (p->container->getSecond())->print();
        cout << " )" << endl;
     }
}

template < class T >
void List<T>::remove(T *q)
{
  listNode<T> *p;
  listNode<T> *t;
  p = head;

  //remove if the first element matches and check the following ones
  while (p && (p->container == q) )
       {
         t = p->next;
         delete p;
         p = t;
       }

  if (!(head = p))
    {              // empty list
      last = 0;
      return;
    }

  //first doesn't matches so check the rest
  for ( t = p; p; p = t->next)
     if (p->container == q)  // match
       {
         t->next = p->next;
         delete p;
       }
     else t = p;
}

#endif
/**************************************************************/
//table.h:  Symbol Table related declarations

#ifndef TABLE
#define TABLE
```

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

class record {
      friend class hashTable;
        char* name;
        char* type;

    public:
        record() :name(0), type(0) {}
        record(char* n, char* t) :name(n), type(t) {}
        ~record() {}
        void putName(char *n) {name = n;}
        void putNameType(char *n) {type = n;}
        char* getName() {return name;}
        char* getNameType() {return type;}
};

struct storageTable {
           record* entry;
           storageTable* next;
};

class hashTable {
       storageTable **table;
       int size;

   public:
       hashTable();
       ~hashTable();
       int hash(char*);
       char* look(char*);    //return type if entry exists
       void insert(record*);
       void print();
};

#endif
/*************************************************************/
// collect.h: specific case definition of Set, Bag or List

#ifndef COLLECT
#define COLLECT

#include <iostream.h>
#include "list.h"

class collect {
       char* type;
       List<char*> clist;
       char* elemtype;
       int number;

  public:
    collect() :type(0), elemtype(0), number(0) {}
    collect(char* t) :type(t), elemtype(0), number(0) {}
    ~collect() {delete type; delete elemtype;}
    void putColType(char* t) {type = t;}
    char* getColType() {return type;}
    void putElemType(char* t) {elemtype = t;}
    char* getElemType() {return elemtype;}
    void putNumber(int i) {number = i;}
    int getNumber() {return number;}
    void appendCollect(char* *t);
    void displayCollect();
    friend ostream& operator << (ostream&, const collect&);
};
```

```
#endif
/*************************************************************/
//lex.h : lexical analyzer class definition

#ifndef LEX
#define LEX
#define BUFSIZE 10

#include "token.h"


class lex {
    token buffer[BUFSIZE];          // token buffer
    int   pos;                      // next free position in buffer


  public:
     lex() : pos(0) {}
     ~lex(){}
     void putbackToken(token);
     token getNextToken();
     int printToken(token);
};

#endif
/*************************************************************/
// nodes.h :  definitions of node classes

#ifndef NODES
#define NODES

#include <string.h>
#include "token.h"
#include "path.h"
```

```
#include "list.h"
#include "collect.h"

class node {
  public:
    node() {}
    ~node(){}
};

class exprNode: public node {

  public:
     exprNode() {}
     ~exprNode(){}
     virtual void print() {}
     virtual void preprint(){}
     virtual quantif getQuantifier() {}
     virtual char* control(){}
};

class identExprNode : public exprNode {
    char* contents;

  public:
    identExprNode(char* i) :contents(i) {}
    ~identExprNode() {}
    void print() {cout << contents;}
    char* control() {return contents;}
};

class intExprNode : public exprNode {
    int contents;
    quantif quantifier;
  public:
    intExprNode(int i)
               :contents(i) {quantifier.quantifierType = colon;
                             quantifier.quantifierNumber = 0;  }
    intExprNode(int i, quantif q) :contents(i), quantifier(q) {}
    ~intExprNode() {}
    quantif getQuantifier() {return quantifier;}
    void print() {cout << contents;}
    char* control() {return "int";}
};

class pathExprNode : public exprNode {
    path contents;
    quantif quantifier;

  public:
    pathExprNode(path i)
                :contents(i){quantifier.quantifierType = colon;
                             quantifier.quantifierNumber = 0;}
    pathExprNode(path i, quantif q)
                :contents(i), quantifier(q) {}
    ~pathExprNode() {}
    quantif getQuantifier() {return quantifier;}
    void print() {cout << contents;}
    char* control() {return contents.getIdentValue();}
};

class collectExprNode : public exprNode {
    collect contents;
    quantif quantifier;

  public:
    collectExprNode(collect i)
                   :contents(i){quantifier.quantifierType = colon;
```

```
                               quantifier.quantifierNumber = 0;}
    collectExprNode(collect i, quantif q)
                   :contents(i), quantifier(q) {}
    ~collectExprNode() {}
    quantif getQuantifier() {return quantifier;}
    void print() {cout << "tmp" << contents.getNumber();}
    void preprint() {contents.displayCollect();}
    char* control() {return "collect";}
};

class binaryExprNode : public exprNode {
    exprNode* first;
    char*     op;
    exprNode* second;

  public:
    binaryExprNode(exprNode* f, char* o, exprNode* s)
                  :first(f), op(o), second(s){}
    ~binaryExprNode() {}
    exprNode* getFirst() {return first;}
    exprNode* getSecond() {return second;}
    char*     getbOp() {return op;}
    void print() {first->print();cout<<" "<<op<<" ";second->print();}
};

class unaryExprNode : public exprNode {
    char*     opu;
    exprNode* operand;

  public:
    unaryExprNode(char* o, exprNode* p) :opu(o), operand(p) {}
    ~unaryExprNode() {}
    exprNode* getOperand() {return operand;}
    char*     getOpu() {return opu;}
```

```
};

class arithExprNode : public binaryExprNode, public unaryExprNode {

  public:
    arithExprNode(exprNode* f, char* o, exprNode* s)
                :binaryExprNode(f, o, s), unaryExprNode(0, 0) {}
    arithExprNode(char* o, exprNode* p)
                :unaryExprNode(o, p), binaryExprNode(0, 0, 0) {}
    ~arithExprNode() {}
};

class boolExprNode : public binaryExprNode, public unaryExprNode {

  public:
    boolExprNode(exprNode* f, char* o, exprNode* s)
                :binaryExprNode(f, o, s), unaryExprNode(0, 0) {}
    boolExprNode(char* o, exprNode* p)
                :unaryExprNode(o, p), binaryExprNode(0, 0, 0) {}
    ~boolExprNode() {}
};

class qualifierNode: public node {
   friend class Manager;
    char*  type;
    exprNode* first;
    char* op;
    exprNode* second;

  public:
     qualifierNode(char* t, exprNode* f, char* o, exprNode* s)
                 :type(t), first(f), op(o), second(s) {}
     ~qualifierNode(){}
    exprNode* getFirst() {return first;}
    exprNode* getSecond(){return second;}
    char*     getOp()    {return op;}
    char* getType() {return type;}
    virtual void updateLocaldefList(qualifierNode*) {}
    virtual void updateFilterList(qualifierNode*)   {}
    virtual int emptyLocaldefList() {}
    virtual int emptyFilterList()   {}
    virtual void displayLocaldefList() {}
    virtual void displayFilterList()   {}
    virtual List<qualifierNode> getLocaldefList() {}
    virtual List<qualifierNode> getFilterList() {}
};

class generatorNode : public qualifierNode {
     List<qualifierNode> localdefList;
     List<qualifierNode> filterList;

  public:
    generatorNode(exprNode* f, exprNode* s)
                 :qualifierNode("generator",f,"<-",s){}
    ~generatorNode() {}
    void updateLocaldefList(qualifierNode *n) {localdefList.append(n);}
    void updateFilterList(qualifierNode *n) {filterList.append(n);}
    int emptyLocaldefList() {return localdefList.empty();}
    int emptyFilterList() {return filterList.empty();}
    void displayLocaldefList() {localdefList.display();}
    void displayFilterList()   {filterList.display();}
    List<qualifierNode> getLocaldefList() {return localdefList;}
    List<qualifierNode> getFilterList() {return filterList;}
};

class localdefNode  : public qualifierNode {
  public:
    localdefNode(exprNode* f, exprNode* s)
```

```
                :qualifierNode("localdef",f,"AS",s){}
    ~localdefNode() {}
};

class filterNode    : public qualifierNode {
  public:
    filterNode(exprNode* f, char* o, exprNode* s)
              :qualifierNode("filter",f,o,s){}
    filterNode(char* o, exprNode* s)
              :qualifierNode("filter", 0, o, s) {}
    ~filterNode() {}
};

class resultNode {
   char* id;
   path  idd;
   char* resultType;

 public:
    resultNode(path i, char* t) :id(0), idd(i), resultType(t) {}
    resultNode(char* i, char* t) :id(i), resultType(t) {}
    ~resultNode()              {}
    char* getId() {return id;}
    path  getIdd()  {return idd;}
    char* getResultType(){return resultType;}

 private:
   resultNode() {}
};
#endif
/***************************************************************/
// Manager.h :  Manager class definition

#ifndef MANAGER
```

```
#define MANAGER

#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include "nodes.h"
#include "lex.h"

extern lex L;

class Manager {
       char* collectionType;
       List<qualifierNode> generatorList;
       resultNode* result;

public:
     Manager() :collectionType(0), result(0){}
     ~Manager() {delete collectionType;
                delete result;
                generatorList.~List();}
     int  parse();
     void translate(int i);
     void codegen(int i);
     void printInsides();
     void printCollectionType();
     void printResult();
     void printPath(path*);
     void printNode(qualifierNode*);
     void printGen(qualifierNode*);
     void printFil(qualifierNode*);
     void printFilter(qualifierNode*);
     void printLocaldef(qualifierNode*);
     void implant(qualifierNode*);
     void printTree();
     void printer(char*);
     qualifierNode* recognizeGLF(exprNode*);

// private:
     char*    recognizeCollectionType();
     int      recognizeResult();
     int      checkpoint(tokenType tT);
     char*    thereAfter();
     char*    recognizeIdentifier();
     path*    recognizePathname();
     path     recognizepathname();
     exprNode* recognizeArithExpr(exprNode*, char*);
     quantif recognizeQuantifier();
     collect declareCollect(char*);
     void     buildDECLARATION(int i);
     void     buildFORSUCHTHAT();
     void     buildRESULT(int i);
};

#endif
```

Appendix B

Translator's files: *.C

```
// main.C :   the program to translate OCL to O++
// to run: translator < inputfile.OCL > outputfile.O++

#include <iostream.h>
#include "Manager.h"
#include "lex.h"
#include "table.h"

lex L;
hashTable symboltable;

int main()
{
  char* majordomo;
  char* steward;
  Manager boss;
  int i = 1;

  for (majordomo = "bienvenue"; majordomo; majordomo = boss.thereAfter())
     {
       steward = majordomo;
       boss.translate(i++);
       boss.~Manager();
     }
   boss.printer(steward);
   return 0;
}
/************************************************************/
// token.C :  lexical tokens

#include <iostream.h>
#include "token.h"

ostream& operator <<(ostream& os, const token& t)
{
  switch(t.type)
  {
    case integer         : os << "integer  "    << t.ivalue; break;
    case identifier      : os << "identifier  " << t.svalue; break;
    case set             : os << "Set"; break;
    case bag             : os << "Bag"; break;
    case list            : os << "List"; break;
    case arrow           : os << "arrow"; break;
    case lessthan        : os << "lessthan"; break;
    case greaterthan     : os << "greaterthan"; break;
    case equal           : os << "equal"; break;
    case lessORequal     : os << "lessORequal"; break;
    case greaterORequal  : os << "greaterORequal"; break;
    case notequal        : os << "notequal"; break;
    case semicolon       : os << "semicolon"; break;
    case bar             : os << "bar"; break;
    case comma           : os << "comma"; break;
    case dot             : os << "dot"; break;
    case lftsqbracket    : os << "lftsqbracket"; break;
    case rgtsqbracket    : os << "rgtsqbracket"; break;
    case lftcurbracket   : os << "lftcurbracket"; break;
    case rgtcurbracket   : os << "rgtcurbracket"; break;
    case lftpar          : os << "lftpar"; break;
    case rgtpar          : os << "lftpar"; break;
    case minus           : os << "minus"; break;
    case colon           : os << "colon"; break;
    case plus            : os << "plus"; break;
    case multiply        : os << "multiply"; break;
    case divide          : os << "divide"; break;
    case endtoken        : os << "endtoken"; break;
    case every           : os << "every"; break;
    case just            : os << "just"; break;
    case atleast         : os << "atleast"; break;
    case atmost          : os << "atmost"; break;
    case reunion         : os << "reunion"; break;
    case differ          : os << "differ"; break;
    case hastype         : os << "hastype"; break;
    case with            : os << "with"; break;
    case and             : os << "and"; break;
    case or              : os << "or"; break;
    case as              : os << "as"; break;
    case some            : os << "some"; break;
    default              : os << "illegal token type"; break;
  };
 return os;
}

/************************************************************/
// path.C :  member function definition of class path

#include "path.h"
#include "table.h"

ostream& operator <<(ostream& os, const path& p)
{
  path temp(p);
  path* th;

  os << temp.getIdentValue();
  th = temp.getNext();
     while (th)
       {
         os << "." << th->getIdentValue();
         th = th->getNext();
       };
   return os;
}

char* path::getPathType()
{                   // consult Symbol table to fetch the type of path
    return "EntityType";  // for now

}

/***********************************************************/
//table.C:  Symbol Table related member functions definitions

#include <stdlib.h>
#include <string.h>
#include "table.h"

#define HASHSIZE 21

hashTable::hashTable()
{
  table = new storageTable* [size = HASHSIZE];
  for (int i = 0; i < size; i++)
     table[i] = 0;
}
```

```
hashTable::~hashTable()
{
  for (int i = 0; i < size; i++)
    {
      storageTable* t;
      storageTable* tt;
      for (t = table[i]; t; t = tt)
        {
          tt = t->next;
          delete t;
        }
    }
  delete table;
}

int hashTable::hash(char *str)
{
  unsigned i;      // to ensure that the hash value is non-negative

  for (i = 0; *str; str++)
     i = *str + 22 * i;

   return i % size;
}

char* hashTable::look(char *str)
{
  for (storageTable* t = table[hash(str)]; t; t = t->next)
     if (!strcmp(str, t->entry->name)) return t->entry->type;

  return 0;
}

void hashTable::insert(record *rec)
```

```
{
  if(look(rec->name)) return; //don't insert - it's already in
                              //insert otherwise:
  int i;
  storageTable* t = new storageTable;
  t->entry = rec;
  t->next = table[i = hash(rec->name)];
  table[i] = t;
}

void hashTable::print()
{
  int i;

  for (i = 0; i <= size; i++)
    for (storageTable* t = table[i]; t; t = t->next)
      cout<<"Id ( "<<t->entry->name<<" ) of Type: "<<t->entry->type<<endl;
}

/**********************************************************/
// collect.C: function definition for collect.h

#include <assert.h>
#include "collect.h"

void collect::appendCollect(char* *t)
{
   clist.append(t);

//  listNode<char*> *p = new listNode<char*>(t);
//  assert(p != 0);
//  if (clist.getHead() == 0) clist.putHead(p);
//  else clist.updateLastNext(p);
//  clist.putLast(p);
}

void collect::displayCollect()
{
  listNode<char*> *p;

  cout << endl << type << "<";
  cout << elemtype << "> tmp";
  cout << number << ";" << endl << endl;

  for ( p = clist.getHead(); p; p = p->getNext())
     {
       cout <<"    tmp"<<number<<".add("<<p->getBucket()<<");"<< endl;
     }
}

ostream& operator <<(ostream& os, const collect& c)
{
  collect temp(c);
  listNode<char*> *p;

  os << temp.type << " [ ";


  for ( p = temp.clist.getHead(); p; p = p->getNext())
       os << " " << p->getBucket() << " ";

  os << " ] " << endl;
}

/***************************************************************/
// lex.C :  lexical analysis of input
#include <iostream.h>
#include <ctype.h>
```

```
#include <stdlib.h>
#include <string.h>
#include "lex.h"
#include "token.h"

#define LENGTH 51     // the max length of an identifier is LENGTH-1

token lex::getNextToken()
{
   char ch, *s, *ss;
   int i=0;

   //check buffer: fetch a token if any
   if (pos) return buffer[--pos];

   do //skip blanks and new lines
   {
      if ( !cin.get(ch) )
         return token(endtoken);
   } while ( isspace(ch) );

   //read in a whole string until blank encountered;
   ss = new char[LENGTH];
   s = ss;
   do
   {
     if (ch == '.')
       {
         if ( !strlen(ss) ) return token(dot);
         cin.putback(ch);
         *s = 0;
         return token(identifier, ss);
       };
     *s++ = ch;
```

```
      if ( !cin.get(ch) ) break;  //get out of this loop if no more chars
    } while ( !isspace(ch) );
    *s = 0;    // end of string read

    if ( isdigit(ss[0]) )       //if meant to be integer
       return token( integer, atoi(ss) );

    while ( i < MAXSTR )
     {
        if ( 0 == strcmp(ss, string[i]) )
            return token(tokenKind[i]);
        i++;
      };
    return token(identifier, ss);
}

void lex::putbackToken(token t)
{
  if (pos >= BUFSIZE) cout << "lex::putbackToken: increase BUFSIZE" << endl;
  else buffer[pos++] = t;
}

int lex::printToken(token t)
{
 cout << "token   < ";
 cout << t ;
 cout << " >" << endl;
 if (t.getTokenType() == endtoken)
    return 0;

 return 1;
}
/**************************************************************/
// Manager.C :  all methods of Manager Class
```

```
#include <iostream.h>
#include <string.h>
#include "Manager.h"
#include "lex.h"
#include "table.h"

extern lex L;
extern hashTable symboltable;

int Manager::parse()  //parse OCL sentence
{
  exprNode* p = 0;

  if (!(collectionType = recognizeCollectionType()))
    {cout<<"Invalid Collection Type"<<endl; return(0);}
  if (!checkpoint(lftsqbracket) ) {cout << "[ missed." << endl; return 0;}
  for ( ; ; )
    {
      implant(recognizeGLF(p));
      if (checkpoint(bar) ) break;
    };
  if (!recognizeResult() )
     {cout << "Invalid Result expression" << endl; return 0;}
  if (!checkpoint(rgtsqbracket) ) {cout << "] missed" << endl; return 0;}
  return(1);
}

void Manager::codegen(int i)   //build O++ sentence
{
   buildDECLARATION(i);
   buildFORSUCHTHAT();
   buildRESULT(i);
}
```

```
void Manager::translate(int i)
{
  if ( !parse() )
     {
        cout << "Parse failed." << endl;
        return;
     }
 // printInsides();     //overall checkup
  codegen(i);
}

int Manager::checkpoint(tokenType tT)
{
  if ( (L.getNextToken()).getTokenType() != tT ) return 0;
  return 1;
}

char* Manager::recognizeCollectionType()
{
   token t;

   switch( (t = L.getNextToken()).getTokenType() )
   {
     case set : return "Set";

     case bag : return "Bag";

     case list: return "List";

     default: L.putbackToken(t);
              return 0;
   };
}
```

```
char* Manager::recognizeIdentifier()
{
   token t;
   char* ident;

   if ((t = L.getNextToken()).getTokenType() != identifier)
     {
      L.putbackToken(t);
      return 0;
     }

   ident = t.getSvalue();
   return ident;
}

path* Manager::recognizePathname()      //recognize a pathname
{
  char* ident;
  path* pathbox;
  path* temp;
  path* nxt;
  token t;

  if (!(ident = recognizeIdentifier())) return 0;

  pathbox = new path(ident);
  temp = pathbox;

  while ((t = L.getNextToken()).getTokenType() == dot)
    {
      temp->putNext(nxt = new path(recognizeIdentifier()));
      temp = nxt;
    };
  L.putbackToken(t);
  return pathbox;
}

path Manager::recognizepathname()      //recognize a pathname
{
  char* ident;
  path pathbox;
  path* temp;
  path* nxt;
  token t;

  if (!(ident = recognizeIdentifier()))
    {
      cout << "Bad Identifier" << endl;
      return 0;
    };
  pathbox.putIdentValue(ident);
  temp = &pathbox;

  while ((t = L.getNextToken()).getTokenType() == dot)
    {
      temp->putNext(nxt = new path(recognizeIdentifier()));
      temp = nxt;
    };

  L.putbackToken(t);
  return pathbox;
}

quantif Manager::recognizeQuantifier()
{
   token t;
```

```
    tokenType tT;
    quantif q;

    q.quantifierType = colon;          //default
    q.quantifierNumber = 0;
    if ( (tT =(t = L.getNextToken()).getTokenType()) == some || tT == every)
      {
      q.quantifierType = tT;
      return q;
      }
    if  (tT == just || tT == atleast || tT == atmost)
      {
      if ((t = L.getNextToken()).getTokenType() != integer)
        {
         cout << "recognizeQuantifier: quantifier illegal integer value" << endl
         return q;
        }
      q.quantifierType = tT;
      q.quantifierNumber = t.getIvalue();
      return q;
      }
    L.putbackToken(t);                // meaning the absence of a quantifier
    return q;
}

collect Manager::declareCollect(char* s)
{
     collect c;
     tokenType tokTp;
     static i = 0;
     char* *tt;
     char* t;

     if (!checkpoint(lftsqbracket) )
```

```
          {
            cout << "recognizeCollect: [ missed." << endl;
            return c;
          }
        c.putColType(s);
        c.putElemType("String");
        c.putNumber(++i);

        cout << endl << c.getColType() << "<";
        cout << c.getElemType() << "> tmp";
        cout << c.getNumber() << ";" << endl << endl;

        for (tokTp = lftsqbracket; tokTp != rgtsqbracket;
                                  tokTp = (L.getNextToken()).getTokenType())
             if (t = recognizeIdentifier())
               {
                 cout <<"    tmp" <<c.getNumber()<< ".add(" <<t<< ");" <<endl;
                 // c.appendCollect(&t);
               }
             else
               {
                 tokTp = (L.getNextToken()).getTokenType();
                 break;
               }
        return c;
}

qualifierNode* Manager::recognizeGLF(exprNode* e)
{
/*------------------------------------------------------------------------*/
/*                                                                        */
/*  This function is to recognize a qualifier, i.e. Generator, Filter    */
/*  or Localdef. It proceeds by recognizing the first component of       */
/*  a qualifier. It should be either a path or an identifier.            */
/*  Since an identifier is a special case of a path name, it makes        */
/*  sense to recognize a path. Then the differentiation is made upon      */
/*  the type of the next token and recognition is done for each type      */
/*  of a qualifier.                                                       */
/*                                                                        */
/*----------------------------------------------------------------------*/
  int i;
  int n;
  char* opr;
  char* sbl;
  path p;
  path pp;
  collect c;
  token t;
  tokenType tokTp;
  quantif quant;
  quantif qquant;
  exprNode* exp;
  exprNode* expp;

  if (!(exp = e))
    {
      quant = recognizeQuantifier();
      p = recognizepathname();
      exp = new pathExprNode(p, quant);
    }

  switch( ( tokTp = (L.getNextToken()).getTokenType() ))
   {
    case arrow: // recognize Generator and update Symbol table
                if (sbl = recognizeCollectionType())
                  {
                   if ( !((c = declareCollect(sbl)).getColType()) )
                     {
```

```
cout << "recognizeGLF: declareCollect(..) returns empty collect object";
                       cout << endl;
                       return 0;
                     }

                   symboltable.insert(new record(p.getIdentValue(),
                                        c.getElemType()));
                   return new generatorNode(new pathExprNode(p),
                                              new collectExprNode(c));
                 }
               pp = recognizepathname();
               symboltable.insert(new record(p.getIdentValue(),
                                              pp.getPathType()));
               return new generatorNode(new pathExprNode(p),
                                         new pathExprNode(pp));
    case as    :  // recognize Localdef
                 return new localdefNode(new pathExprNode(p),
                                          new pathExprNode(recognizepathname()))
    case equal:
    case lessthan:
    case greaterthan:
    case lessORequal:
    case greaterORequal:
    case notequal:
    case hastype:
                 for(i=0; ;i++)
                   if ( tokTp == tokenKind[i] )
                      { opr = string[i];
                        break;
                      }
              // if ( tokTp == hastype ) opr = "is";
                 if ((t = L.getNextToken()).getTokenType() == integer)
                   return new filterNode(exp, opr,
                          recognizeArithExpr(new intExprNode(t.getIvalue()),0))
```

```
                L.putbackToken(t);
                qquant = recognizeQuantifier();
                if ((t = L.getNextToken()).getTokenType() == identifier)
                  {
                    L.putbackToken(t);
                    return new filterNode(exp, opr,
                         recognizeArithExpr(new pathExprNode(pp =
                                          recognizepathname(), qquant), 0));
                  }
                if ((t.getTokenType() == set) || (t.getTokenType() == bag) ||
                                               (t.getTokenType() == list))
                  {
                    L.putbackToken(t);
                    return new filterNode(exp, opr,
                           new collectExprNode(declareCollect(
                              recognizeCollectionType()), qquant));
                  }
   case minus:       /****************************************/
   case plus:        /*   recognize arithmetic expression    */
   case multiply:    /****************************************/
   case divide:   for(i=0; ;i++)
                  if ( tokTp == tokenKind[i] )
                    { opr = string[i];
                      break;
                    }
                  return recognizeGLF(recognizeArithExpr(exp, opr));

   default:       // just in case..
                  cout << "recognizeGLF: default was called" << endl;
                  return 0;
  };
}
```

```
exprNode* Manager::recognizeArithExpr(exprNode* p, char* oper)
{
  path *pp;
  token t;
  char* opr;
  int i;
  tokenType tokTp;
  exprNode* en;
  binaryExprNode* ben;

  if (!(opr = oper))
    {                              // no operation specified; go and probe
     switch (tokTp = (t = L.getNextToken()).getTokenType())
      {
       case minus:
       case plus:
       case multiply:
       case divide:
                   for(i=0; ;i++)
                      if ( tokTp == tokenKind[i] )
                        {
                         opr = string[i];
                         break;
                        }
                // return recognizeArithExpr(p, opr);
                   break;

       default:  // end of ArithExpr or not ArithExpr kind
                 L.putbackToken(t);
                 return p;
      }
    }

  if (!(pp = recognizePathname()))
```

```
    {
      if ((t = L.getNextToken()).getTokenType() == integer)
        return recognizeArithExpr(
               (binaryExprNode*) new arithExprNode(p, opr,
                                  new intExprNode(t.getIvalue())), 0);
    }
  else return recognizeArithExpr(
              (binaryExprNode*) new arithExprNode(p, opr,
                                 new pathExprNode(*pp)), 0);
}

int Manager::recognizeResult()
{
   path reslt;
   char* type;

   reslt = recognizepathname();
   if (!(type = reslt.getPathType()))
     {
      cout<<"recognizeResult: check Result Type (not in SymbolTable?)"<<endl;
      return 0;
     }
   result = new resultNode(reslt, type);
   return 1;
}

void Manager::buildDECLARATION(int i)
{
  cout << endl << collectionType;
  cout <<"<"<<result->getResultType()<<"> temp"<<i<<";"<<endl<<endl;
}

void Manager::buildFORSUCHTHAT()
{
```

```
listNode<qualifierNode> *g;
listNode<qualifierNode> *d;
listNode<qualifierNode> *f;
List<qualifierNode> assistantLocaldefList;
qualifierNode* n;
exprNode* etalon;
char* filtflag;
char* printer;

for ( g = generatorList.getHead(); g; g = g->getNext() )
   {
    printGen( n = g->getContainer() );
    filtflag = "suchthat";
    assistantLocaldefList = n->getLocaldefList();
    for (f = (n->getFilterList()).getHead(); f; f = f->getNext() )
      {
        if (filtflag) {cout << "    suchthat ( "; filtflag = 0;}
        else cout << " && ";
        printer = 0;
        etalon = (f->getContainer())->getFirst();

        // in case localdef list is NOT empty
        for ( d = assistantLocaldefList.getHead(); d; d = d->getNext() )
          if (strcmp(((d->getContainer())->getFirst())->control(),
                      etalon->control()) == 0)
           {
             cout << "(";
             printLocaldef(d->getContainer());
             printFil(f->getContainer());      //printing the rest of filter
             printer = "busy";
             assistantLocaldefList.remove(d->getContainer());
             break;
            }
            if (!printer) printFilter(f->getContainer() );
          }
        if (!filtflag) cout << " )" << endl;
        cout << "    ";
      }
   }

   void Manager::buildRESULT(int i)
   {
     cout <<"    temp"<< i << ".add(" <<result->getIdd()<<");"<<endl<<endl;
   }

   void Manager::printCollectionType()
   {
       cout << "|------------------------------------------------|" << endl;
       cout << "|                                                |" << endl;
       cout << "| - collectionType (" << collectionType << ")" << endl;
   }

   void Manager::printResult()
   {
       cout << "| - result    ( "<<result->getIdd()<<" ) of Type "<<
                    result->getResultType()<<endl;
       cout << "|                                                |" << endl;
       cout << "|------------------------------------------------|" << endl;
   }

   void Manager::printPath(path* p)
   {
     path* temp;

       temp = p;
      cout << "pathname <" << temp->getIdentValue();
      while(temp = temp->getNext())
   {
     cout << "." << temp->getIdentValue();
   }
   cout << ">" << endl;
}

void Manager::printNode(qualifierNode* n)
{
   cout << n->getType() << "  ( ";
   (n->getFirst())->print();
   cout << " " << n->getOp() << " ";
   (n->getSecond())->print();
   cout << " )" << endl;
}

void Manager::printGen(qualifierNode* n)
{
  // if (strcmp((n->getSecond())->control(), "collect") == 0)
  //    (n->getSecond())->preprint();

   cout << "for (";
   (n->getFirst())->print();
   cout << " in ";
   (n->getSecond())->print();
   cout << ")" << endl;
}

void Manager::printFil(qualifierNode* n)   // partial printing of a filter
{
   if (n->getOp())
     cout << " " << n->getOp() << " ";
   (n->getSecond())->print();
   cout << ")";
}
```

```
void Manager::printFilter(qualifierNode* n)  // full printing of a filter
{
   quantif quantFirst;
   quantif quantSecond;
   cout << "(";

// check for quantifiers and print accordingly
   quantFirst = n->getFirst()->getQuantifier();
   quantSecond = n->getSecond()->getQuantifier();
   if (quantFirst.quantifierType == some)
      switch (quantSecond.quantifierType)
         {
          case some:     n->getFirst()->print();
                         cout << ".intersection(";
                         n->getSecond()->print();
                         cout << ")";
                         return;

          case atleast: n->getSecond()->print();
                         cout << ".atleast(";
                         cout << quantSecond.quantifierNumber << ", ";
                         n->getFirst()->print();
                         cout << ")";
                         return;

          case just:     n->getSecond()->print();
                         cout << ".just(";
                         cout << quantSecond.quantifierNumber << ", ";
                         n->getFirst()->print();
                         cout << ")";
                         return;

          case atmost:  n->getSecond()->print();
```

```
                         cout << ".atmost(";
                         cout << quantSecond.quantifierNumber << ", ";
                         n->getFirst()->print();
                         cout << ")";
                         return;

           default:      cout << endl;
                         cout <<"printFilter: quantifier has an illegal value";
                         cout << endl;
                         return;
          }
      if ((quantFirst.quantifierType == every) &&
          (quantSecond.quantifierType == some))
        {
          n->getFirst()->print();
          cout << ".every(";
          n->getSecond()->print();
          cout << ")";
          return;
        }

      if ((quantFirst.quantifierType == colon) &&
          (quantSecond.quantifierType == some))
        {
          n->getSecond()->print();
          cout << ".member(";
          n->getFirst()->print();
          cout << ")";
          return;
        }

      if ((quantFirst.quantifierType == just) &&
          (quantSecond.quantifierType == colon))
        {
```

```
      n->getFirst()->print();
      cout << ".frequency(";
      n->getSecond()->print();
      cout << ") == " << quantFirst.quantifierNumber;
      return;
     }
//there are no quantifiers - proceed with the "simple" printing
   n->getFirst()->print();
   printFil(n);
}

void Manager::printLocaldef(qualifierNode* n)
{
   cout << "(";
   (n->getFirst())->print();
   cout << " = ";
   (n->getSecond())->print();
   cout << ")";
}

void Manager::printer(char *s)
{
  if (strcmp(s, "bienvenue") )
  cout << "temp1." << s << "(temp2);" << endl;
}

void Manager::printTree()
{
//  cout << "Generator List:"<<endl;
//  generatorList.display();

  listNode<qualifierNode> *p;
  listNode<qualifierNode> *d;
  qualifierNode* n;
  cout << endl << "Query Tree:"<< endl << endl;
  for ( p = generatorList.getHead(); p; p = p->getNext() )
     {
      printNode( n = p->getContainer() );

      for ( d = (n->getLocaldefList()).getHead(); d; d = d->getNext() )
         {
           cout << "     ";
           printNode( d->getContainer() );
         }

      for ( d = (n->getFilterList()).getHead(); d; d = d->getNext() )
         {
           cout << "          ";
           printNode( d->getContainer() );
         }
      cout << endl;
     }
}

void Manager::implant(qualifierNode *n)
{
 if (n->getType() == "generator")
   {
    generatorList.append(n);
    return;
   }

 if (n->getType() == "localdef")
  {
/*-----------------------------------------------------------------------*/
/*                                                                       */
/*  since this localdef belongs to the last parsed generator, go and get */
/*  the last implanted generatorNode, then append this localdef to the   */
/*  localdefList of the generatorNode.                                   */
/*                                                                       */
/*-----------------------------------------------------------------------*/
   ((generatorList.getLast())->getContainer())->updateLocaldefList(n);
   return;
  }

 if (n->getType() == "filter")
  {
/*-----------------------------------------------------------------------*/
/*                                                                       */
/*  since this filter belongs to the last parsed generator, go and get   */
/*  the last implanted generatorNode, then append this filter to the     */
/*  filterfList of the generatorNode.                                    */
/*                                                                       */
/*-----------------------------------------------------------------------*/
   ((generatorList.getLast())->getContainer())->updateFilterList(n);
   return;
  }
/*-----------------------------------------------------------------------*/
/*                                                                       */
/* if it is not a generator and neither a localdef, nor a filter,        */
/* then it's illegal qualifier type. An error message is to be issued.   */
/*                                                                       */
/*-----------------------------------------------------------------------*/
 cout << "implant: Illegal qualifier type" << endl;  //just in case..
}

void Manager::printInsides()
{
   printCollectionType();
   printTree();
   printResult();
   cout << endl << "Symbol Table Contents:" << endl;
   symboltable.print();
}

char* Manager::thereAfter()
{
  switch ( (L.getNextToken()).getTokenType() )
   {
     case reunion:  return "union";

     case differ:   return "differ";

     case endtoken: return 0;

     default: cout << "thereAfter: something wrong in here" << endl;
              return 0;
   }
}
```

Appendix C

Utility code

This appendix contains code of `collection.h`, `strg.h` and `collection.c`.

```
//collection.h

template <class Type>
class collection {

protected:
    node<Type> *head;
    int  total;

public:
  collection() {total = 0; head = 0;}
  ~collection() {destroy();}

  virtual Boolean member(Type);
  virtual int size() {return total;}
  virtual void remove(Type);
  virtual Boolean intersects(collection<Type>*);
  virtual void intersection(collection<Type>*);
  virtual Boolean every(collection<Type>*);
  virtual void union_col(collection<Type>*);
  virtual Boolean atleast(int,collection<Type>*);
  virtual Boolean atmost(int,collection<Type>*);
  virtual Boolean just(int,collection<Type>*);
  virtual void differ(collection<Type>*);
  virtual void destroy(); // remove all items
  virtual void display();
  virtual Boolean is_empty() = 0;
  virtual void insert(Type,int) = 0;
  virtual void append(Type) = 0;
  virtual int frequency(Type) = 0;
};
/* ============================================================ */

template <class Type>
class list : public collection<Type> {

public:
  list()  : collection<Type>() {};
  Boolean is_empty();
  void remove(Type v) {del(findnb(v));}
  void insert(Type, int);
  void append(Type v) {insert(v, total+1);}
  int findnb(Type);
  void modify(Type, int);
  int  frequency(Type);
  Type show(int);
  void del(int);
  list<Type>* isublist(int,int);
  list<Type>* sublist(Type j, Type k) {return isublist(findnb(j), findnb(k));}
};
/* ============================================================ */

template <class Type>
class bag : public collection<Type> {
```

```
public:
  bag() : collection<Type>() {};
  void add(const Type&);
  int frequency(Type);
};
/* ================================================================= */

template <class Type>
class set :  public collection<Type> {

public:
  set() : collection<Type>() {};
  void add(const Type&);
  void union_set(set<Type>*);
};
/* ================================================================= */

template <class Type>
class creator {

public:
    creator();
    collection<Type>* create() {return new list<Type>;}
};
/* ================================================================= */

/* template <class Type>
collection<Type>* creator<Type>::create(char* ds)
{
list<Type> *plist;
collection<Type> *pclist;

  if (!strcmp(ds,"list"))
    {
      plist = new list<Type>;
      pclist = plist;
    }

  return pclist;

}; */
/* ==================================================================== */

template<class Type>
Boolean list<Type>::is_empty()
{
 return total == 0 ? true : false;
};
/* ==================================================================== */

template<class Type>
Boolean collection<Type>::member(Type val)
{
if (!head)
   return false;

node<Type> *cursor = head;

while (cursor) {
   if (cursor->isequal(cursor->get_info(), val))
        return true;
   cursor = cursor->get_next();
  }
return false;
};
/* ==================================================================== */

template<class Type>
```

```
void collection<Type>::destroy()
{
  node<Type> *cursor = head, *tmp;

  while(cursor) {


      tmp = cursor;
      cursor = cursor -> get_next();
      delete tmp;
  }
  head = 0;
  total = 0;
};
/* ================================================================== */

template<class Type>
void list<Type>::del(int pos)
{
 node<Type> *prev, *cursor = head;
 if (pos == 1) {
    head = cursor->get_next();
    delete cursor;
    total--;
    if (total == 0)
        head = 0;
   }
 else
     if (pos > 1 && pos <= total)
    {
     for(int i = 1; (i < pos) ; i++) {
        prev = cursor;
        cursor = cursor->get_next();
     }
     prev->put_next(cursor->get_next());
     delete cursor;
     total--;
    }
};
/* ================================================================== */

template<class Type>
void list<Type>::insert(Type val, int pos)
{
 node<Type> *prev, *cursor = head, *temp = new node<Type>(val);

 if (!head) {
     head = temp;
     total++;
     }
 else
     if (pos == 1) { // insert at head
         temp->put_next(head);
         head = temp;
         total++;
     }
     else
           if (pos > 1)
          {
           for(int i = 1; (i < pos) && cursor->get_next(); i++) {
              prev = cursor;
              cursor = cursor->get_next();
           }

           if (!cursor->get_next() && (i < pos))
              cursor->put_next(temp);
           else {
                 temp->put_next(cursor);
                 prev->put_next(temp);
                 }
            total++;
           }
};
/* ================================================================== */

template<class Type>
int list<Type>::findnb(Type val)
{
 int cnt = 0;

 if (!head)
    return cnt;

 node<Type> *cursor = head;

 while(cursor) {
    cnt++;
    if (cursor->isequal(cursor->get_info(),val))
       return cnt;
    cursor = cursor->get_next();
 }

 return 0;
};
/* ================================================================== */

template<class Type>
int list<Type>::frequency(Type val)
{
 node<Type> *cursor = head;
 int count = 0;

 if (!head)
```

```
     cout << "List is empty!" << endl;
 else {
      while (cursor) {
           if (cursor->isequal(cursor->get_info(),val))
                count++;
      cursor = cursor->get_next();
      }
  }
  return count;
};
/* ================================================================= */

template<class Type>
Type list<Type>::show(int pos)
{

 node<Type> *cursor = head;

 if ((pos <= total) && (cursor))
 {
  for (int i=1; i < pos;i++)
      cursor = cursor->get_next();

  return(cursor->get_info());
 }

 return 0;
};
/* ================================================================= */

template<class Type>
list<Type>* list<Type>::isublist(int min, int max)
{
  list<Type> *lst = new list<Type>;
  node<Type> *cursor = head;

  if ((!head) || (min > max))
     return lst;

  for (int i=1; (i <= max) && cursor; i++){
      if ((i >= min) && (i <= max))
         lst->append(cursor->get_info());
      cursor = cursor->get_next();
     }
  return lst;
 };
 /* ================================================================= */

 template <class Type>
 void collection<Type>::intersection(collection<Type> *lst)
 {
  node<Type> *crs1 = head, *temp, *prev = head; // current one
  node<Type> *crs2 = lst->head;
  int flag = 1;

  while(crs1) {
   while (crs2 && flag) {
   if (crs1->isequal(crs1->get_info(), crs2->get_info()))
       flag = 0;
   crs2 = crs2->get_next();
  }

   if (flag){
        prev->put_next(crs1->get_next());
        temp = crs1;
        crs1 = crs1->get_next();
        if (temp == head)
```

```
            head = crs1;
        delete temp;
        total--;
        if (total == 0)
           head = 0;
        }
  else
       {
        prev = crs1;
        crs1 = crs1->get_next();
       }

  crs2 = lst->head;
  flag = 1;
 }
};
/* ==================================================================== */

template <class Type>
Boolean collection<Type>::intersects(collection<Type> *lst)
{
 node<Type> *cursor1 = head;
 node<Type> *crs2 = lst->head;

 while(cursor1) {
  while (crs2) {
  if (cursor1->isequal(cursor1->get_info(), crs2->get_info()))
      return true;
  crs2 = crs2->get_next();
 }
  cursor1 = cursor1->get_next();
  crs2 = lst->head;
 }
return false;
```

```
};
/* ================================================================= */

template <class Type>
void collection<Type>::differ(collection<Type> *lst)
{
 node<Type> *crs1 = head, *temp, *prev = head; // current one
 node<Type> *crs2 = lst->head;
 int flag = 1;

 while(crs1) {
  while (crs2 && flag) {
  if (crs1->isequal(crs1->get_info(), crs2->get_info()))
      flag = 0;
  crs2 = crs2->get_next();
 }

  if (!flag){
       prev->put_next(crs1->get_next());
       temp = crs1;
       crs1 = crs1->get_next();
       if (temp == head && crs1)
           head = crs1;
       delete temp;
       total--;
       if (total == 0)
          head = 0;
       }
  else
      {
       prev = crs1;
       crs1 = crs1->get_next();
      }
  crs2 = lst->head;
  flag = 1;
 }
};
/* ================================================================== */

template <class Type>
void collection<Type>::union_col(collection<Type> *lst)
{
 node<Type> *cursor1 = head;  /* current list */
 node<Type> *cursor2 = lst->head;


 while(cursor1->get_next())
    cursor1 = cursor1->get_next();

 cursor1->put_next(cursor2);
 total = total + lst->total;
};
/* ================================================================== */

template <class Type>
void set<Type>::union_set(set<Type> *st)
{
 node<Type> *cursor1 = head, *prev = head;  /* current set */
 node<Type> *crs2 = st->head;
 int flag = 1;

 while (crs2) {
   while(cursor1 && flag) {
      if (crs2->isequal(cursor1->get_info(), crs2->get_info()))
          flag = 0;
      else
          prev = cursor1;
          cursor1 = cursor1->get_next();
   }

  if (flag) {
      prev->put_next(crs2);
      total++;
  }

  cursor1 = head;
  prev = head;
  crs2 = crs2->get_next();
  flag = 1;

}
};
/* ================================================================== */

template <class Type>
Boolean collection<Type>::every(collection<Type> *lst)
{
 node<Type> *cursor1 = head;  /* current list */
 node<Type> *crs2 = lst->head;

 int flag = 1;

 while (cursor1){
   while (crs2 && flag) {
     if (cursor1->isequal(cursor1->get_info(), crs2->get_info()))
        flag = 0;
     else
        crs2 = crs2->get_next();
   }
  if (flag)
     return false;
  else {
       cursor1 = cursor1->get_next();
       crs2 = lst->head;
       flag = 1;
      }
 }
return true;
};
/* ================================================================== */

template <class Type>
Boolean collection<Type>::atleast(int al, collection<Type> *lst)
{

 node<Type> *cursor1 = head;  /* current list */
 node<Type> *crs2 = lst->head;

 int flag = 1;
 int cnt = 0;

 while (cursor1){
   while (crs2 && flag) {
     if (cursor1->isequal(cursor1->get_info(), crs2->get_info())) {
        cnt++;
        flag = 0; }
     else
        crs2 = crs2->get_next();
   }
   cursor1 = cursor1->get_next();
   crs2 = lst->head;
   flag = 1;

}
   if (cnt >= al)
```

```
        return true;
    else
        return false;
};
/* ================================================================ */

template <class Type>
Boolean collection<Type>::atmost(int am, collection<Type> *lst)
{

 node<Type> *cursor1 = head;  /* current list */
 node<Type> *crs2 = lst->head;

 int flag = 1;
 int cnt = 0;

 while (cursor1){
   while (crs2 && flag) {
     if (cursor1->isequal(cursor1->get_info(), crs2->get_info())) {
        cnt++;
        flag = 0; }
     else
        crs2 = crs2->get_next();
   }
   cursor1 = cursor1->get_next();
   crs2 = lst->head;
   flag = 1;

}
   if (cnt <= am)
       return true;
   else
       return false;
};
```

```
/* ================================================================== */

template <class Type>
Boolean collection<Type>::just(int js, collection<Type> *lst)
{

 node<Type> *cursor1 = head;  /* current list */
 node<Type> *crs2 = lst->head;

 int flag = 1;
 int cnt = 0;

 while (cursor1){
   while (crs2 && flag) {
     if (cursor1->isequal(cursor1->get_info(), crs2->get_info())) {
        cnt++;
        flag = 0; }
     else
        crs2 = crs2->get_next();
   }
   cursor1 = cursor1->get_next();
   crs2 = lst->head;
   flag = 1;

}
   if (cnt == js)
       return true;
   else
       return false;
};
/* ================================================================== */

template <class Type>
void collection<Type>::display()
```

```
{
 node<Type> *cursor = head;

 Type temp;

 while (cursor) {
    temp = cursor->get_info();
    cout << "*" << temp.retstr() << endl;
    cursor = cursor->get_next();
 }
};
/* ================================================================== */

template <class Type>
void collection<Type>::remove(Type info)
{
 node<Type> *prev = head, *cursor = head, *temp;
 Type tep = info;
 int flag = 1;

 while (cursor && flag) {
   tep = cursor->get_info();
   if (!(cursor->isequal(tep,info))){
      prev = cursor;
      cursor = cursor->get_next();
     }
   else
     {
      prev->put_next(cursor->get_next());
      temp = cursor;
      cursor = cursor->get_next();
      if (temp == head && cursor)
          head = cursor;
      delete temp;
```

```
        total--;
        if (total == 0)
           head = 0;
        flag = 0;
       }
 }
};
/* ================================================================= */

template <class Type>
void bag<Type>::add(const Type &info)
{
 node<Type> *cursor = head, *crs = head;

 node<Type> *temp = new node<Type>(info);

 if (!head) {
     head = temp;
 }
 else
     {
       while (cursor->get_next()) {
              cursor = cursor->get_next();
       }
       cursor->put_next(temp);

    }
 total++;
};
/* ================================================================= */

template <class Type>
int bag<Type>::frequency(Type info)
{
 node<Type> *cursor = head;
 Type temp = info;
 int count = 0;

 if (!head)
     cout << "List is empty!" << endl;
 else
  {
      while (cursor)
      {
          temp = cursor->get_info();
          if (cursor->isequal(temp,info))
              count++;
      cursor = cursor->get_next();
      }
  }
  return count;
};
/* ================================================================= */

template <class Type>
void set<Type>::add(const Type &info)
{
 node<Type> *cursor = head, *crs = head;

 node<Type> *temp = new node<Type>(info);

 int flag = 1;

 if (!head) {
     head = temp;
     total++;
 }
 else
```

```
     {
       while (cursor->get_next() && flag) {
               if (cursor->isequal(cursor->get_info(),info))
                   flag = 0;
               cursor = cursor->get_next();
       }
       if (flag) {
          cursor->put_next(temp);
          total++;
         }
    }
};
/* ================================================================ */

//strg.h

class String {
    int len;
    char *str;

public:
    String() {len = 0; str = (char*)0;}
    String(char*);
    char* retstr() {return str;}
};

String::String(char *s)
{
 len = strlen(s);
 str = new char[len+1];
 assert(str != 0);
 strcpy(str,s);
};
/* ================================================================ */
```

```
//collection.c

#include <string.h>
#include <assert.h>
#include "/mnt/gb1/Oodbms/strg.h"
#include "/mnt/gb1/Oodbms/collection.h"



template<class Type>
Boolean list<Type>::is_empty()
{
 return total == 0 ? true : false;
};
/* ================================================================== */

template<class Type>
Boolean collection<Type>::member(Type val)
{
if (!head)
   return false;

node<Type> *cursor = head;

while (cursor) {
   if (cursor->isequal(cursor->get_info(), val))
        return true;
   cursor = cursor->get_next();
  }
return false;
};
/* ================================================================== */

template<class Type>
```

```
void collection<Type>::destroy()
{
  node<Type> *cursor = head, *tmp;

  while(cursor) {

      tmp = cursor;
      cursor = cursor -> get_next();
      delete tmp;
  }
  head = 0;
  total = 0;
};
/* ================================================================== */

template<class Type>
void list<Type>::del(int pos)
{
 node<Type> *prev, *cursor = head;
 if (pos == 1) {
    head = cursor->get_next();
    delete cursor;
    total--;
    if (total == 0)
        head = 0;
   }
 else
     if (pos > 1 && pos <= total)
    {
     for(int i = 1; (i < pos) ; i++) {
        prev = cursor;
        cursor = cursor->get_next();
     }
     prev->put_next(cursor->get_next());
     delete cursor;
     total--;
    }
};
/* ===================================================================== */

template<class Type>
void list<Type>::insert(Type val, int pos)
{
 node<Type> *prev, *cursor = head, *temp = new node<Type>(val);

 if (!head) {
     head = temp;
     total++;
     }
 else
     if (pos == 1) { // insert at head
         temp->put_next(head);
         head = temp;
         total++;
     }
     else
           if (pos > 1)
          {
           for(int i = 1; (i < pos) && cursor->get_next(); i++) {
              prev = cursor;
              cursor = cursor->get_next();
           }

           if (!cursor->get_next() && (i < pos))
              cursor->put_next(temp);
           else {
                  temp->put_next(cursor);
                  prev->put_next(temp);
                  }
              total++;
            }
};
/* ==================================================================== */

template<class Type>
int list<Type>::findnb(Type val)
{
 int cnt = 0;

 if (!head)
    return cnt;

 node<Type> *cursor = head;

 while(cursor) {
    cnt++;
    if (cursor->isequal(cursor->get_info(),val))
       return cnt;
    cursor = cursor->get_next();
 }

 return 0;
};
/* ==================================================================== */

template<class Type>
int list<Type>::frequency(Type val)
{
 node<Type> *cursor = head;
 int count = 0;

 if (!head)
```

```
     cout << "List is empty!" << endl;
 else {
      while (cursor) {
          if (cursor->isequal(cursor->get_info(),val))
              count++;
      cursor = cursor->get_next();
      }
  }
  return count;
};
/* ================================================================== */

template<class Type>
Type list<Type>::show(int pos)
{

 node<Type> *cursor = head;

 if ((pos <= total) && (cursor))
 {
  for (int i=1; i < pos;i++)
      cursor = cursor->get_next();

  return(cursor->get_info());
 }
 return 0;
};
/* ================================================================== */

template<class Type>
list<Type>* list<Type>::isublist(int min, int max)
{

 list<Type> *lst = new list<Type>;
 node<Type> *cursor = head;

 if ((!head) || (min > max))
    return lst;

 for (int i=1; (i <= max) && cursor; i++){
     if ((i >= min) && (i <= max))
        lst->append(cursor->get_info());
     cursor = cursor->get_next();
    }
 return lst;
};
/* ================================================================== */

template <class Type>
void collection<Type>::intersection(collection<Type> *lst)
{
 node<Type> *crs1 = head, *temp, *prev = head; // current one
 node<Type> *crs2 = lst->head;
 int flag = 1;

 while(crs1) {
  while (crs2 && flag) {
  if (crs1->isequal(crs1->get_info(), crs2->get_info()))
      flag = 0;
  crs2 = crs2->get_next();
 }

  if (flag){
       prev->put_next(crs1->get_next());
       temp = crs1;
       crs1 = crs1->get_next();
       if (temp == head)
           head = crs1;
        delete temp;
        total--;
        if (total == 0)
           head = 0;
        }
   else
       {
        prev = crs1;
        crs1 = crs1->get_next();
       }

   crs2 = lst->head;
   flag = 1;
 }
};
/* ================================================================= */

template <class Type>
Boolean collection<Type>::intersects(collection<Type> *lst)
{
 node<Type> *cursor1 = head;
 node<Type> *crs2 = lst->head;

 while(cursor1) {
  while (crs2) {
  if (cursor1->isequal(cursor1->get_info(), crs2->get_info()))
      return true;
  crs2 = crs2->get_next();
 }
  cursor1 = cursor1->get_next();
  crs2 = lst->head;
 }
return false;
};
```

```
/* ================================================================== */

template <class Type>
void collection<Type>::differ(collection<Type> *lst)
{
 node<Type> *crs1 = head, *temp, *prev = head; // current one
 node<Type> *crs2 = lst->head;
 int flag = 1;

 while(crs1) {
  while (crs2 && flag) {
  if (crs1->isequal(crs1->get_info(), crs2->get_info()))
      flag = 0;
  crs2 = crs2->get_next();
 }
  if (!flag){
       prev->put_next(crs1->get_next());
       temp = crs1;
       crs1 = crs1->get_next();
       if (temp == head && crs1)
           head = crs1;
       delete temp;
       total--;
       if (total == 0)
          head = 0;
       }
  else
      {
       prev = crs1;
       crs1 = crs1->get_next();
      }
  crs2 = lst->head;
  flag = 1;
 }
```

```
};
/* ============================================================ */

template <class Type>
void collection<Type>::union_col(collection<Type> *lst)
{
 node<Type> *cursor1 = head;  /* current list */
 node<Type> *cursor2 = lst->head;


 while(cursor1->get_next())
    cursor1 = cursor1->get_next();

 cursor1->put_next(cursor2);
 total = total + lst->total;
};
/* ============================================================ */

template <class Type>
Boolean collection<Type>::every(collection<Type> *lst)
{
 node<Type> *cursor1 = head;  /* current list */
 node<Type> *crs2 = lst->head;

 int flag = 1;
 while (cursor1){
   while (crs2 && flag) {
     if (cursor1->isequal(cursor1->get_info(), crs2->get_info()))
        flag = 0;
     else
        crs2 = crs2->get_next();
   }
  if (flag)
     return false;
   else {
        cursor1 = cursor1->get_next();
        crs2 = lst->head;
        flag = 1;
       }
 }
return true;
};
/* ================================================================== */

template <class Type>
Boolean collection<Type>::atleast(int al, collection<Type> *lst)
{

 node<Type> *cursor1 = head;  /* current list */
 node<Type> *crs2 = lst->head;

 int flag = 1;
 int cnt = 0;

 while (cursor1){
   while (crs2 && flag) {
     if (cursor1->isequal(cursor1->get_info(), crs2->get_info())) {
        cnt++;
        flag = 0; }
     else
        crs2 = crs2->get_next();
   }
   cursor1 = cursor1->get_next();
   crs2 = lst->head;
   flag = 1;

}
   if (cnt >= al)
```

```
        return true;
    else
        return false;
};
/* ================================================================= */

template <class Type>
Boolean collection<Type>::atmost(int am, collection<Type> *lst)
{

 node<Type> *cursor1 = head;  /* current list */
 node<Type> *crs2 = lst->head;

 int flag = 1;
 int cnt = 0;

 while (cursor1){
   while (crs2 && flag) {
     if (cursor1->isequal(cursor1->get_info(), crs2->get_info())) {
        cnt++;
        flag = 0; }
     else
        crs2 = crs2->get_next();
   }
   cursor1 = cursor1->get_next();
   crs2 = lst->head;
   flag = 1;

}
   if (cnt <= am)
       return true;
   else
       return false;
};
```

```
/* ================================================================= */

template <class Type>
Boolean collection<Type>::just(int js, collection<Type> *lst)
{

 node<Type> *cursor1 = head;  /* current list */
 node<Type> *crs2 = lst->head;

 int flag = 1;
 int cnt = 0;

 while (cursor1){
   while (crs2 && flag) {
     if (cursor1->isequal(cursor1->get_info(), crs2->get_info())) {
        cnt++;
        flag = 0; }
     else
        crs2 = crs2->get_next();
   }
   cursor1 = cursor1->get_next();
   crs2 = lst->head;
   flag = 1;

}
   if (cnt == js)
       return true;
   else
       return false;
};
/* ================================================================= */

template <class Type>
void collection<Type>::display()
```

```
{
 node<Type> *cursor = head;

 Type temp;

 while (cursor) {
    temp = cursor->get_info();
    cout << "*" << temp.retstr() << endl;
    cursor = cursor->get_next();
 }
};
```