Query Translation in Heterogeneous Database Environments

Edgar Jasper

MSc Computing Science project report,
School of Computer Science and Information Systems,
Birkbeck College,
University of London,
2001

CONTENTS

ABSTRACT

One of the things that must be achieved to implement a heterogeneous database system is global query processing. Global queries expressed on the global schema must be translated into queries that can be processed by the local databases. Using a low-level graph-based data model as the common data model this project investigates the automatic translation of such global queries. This automatic translation is accomplished by using the transformation pathways from the local schemas to the global schema.

The goal of this project is to produce software to translate global queries into queries expressed in terms of the local database constructs. A secondary goal is to produce software to optimise these translated queries for faster execution.

1. AIMS AND BACKGROUND

1.1 General Background

It may be desirable to integrate a number of autonomous databases in such a way that to the user, be that a person or another layer of software, there appears to be a single database. This single ‘multi-database’ would sit on top of the autonomous databases - databases that may employ different data models and be deployed on a variety of platforms. This system is a heterogeneous database system.

There are four main things to achieve to implement such a system. These are schema translation, schema integration, global query processing (including optimisation), and global transaction management. Obviously, given that the local databases may employ different data models, the local schemas must be translated into component schemas (see [7]) expressed in some suitable common data model (CDM) so that they may be integrated to produce the global schema expressed also in this CDM. The need for such schema translation and integration, while not the subject of this project, has bearing on it and will be discussed more later on.

Global query processing consists, in brief, of translating a global query expressed on the global schema into one in terms of the component schemas from which the global schema was derived. The resulting query is then optimised. Then the local sub-queries expressed in the global query language of the CDM are translated into queries expressed in local query languages appropriate for the local schemas. Finally these local sub-queries are sent out for processing by the local databases. This project is concerned with an aspect of this global query processing.

This project is not concerned with global transaction management so it will not be discussed further.

1.2 Specific Background

This project investigates global query processing. It is a part of the AutoMed project at Birkbeck College and Imperial College and as such works within the following framework for schema integration and query processing developed by P. McBrien and A. Poulovassilis. The CDM used by this framework is the hypergraph data model (see [10]). This model is a low level model where schemas consist of nodes, edges (directed edges that can link multiple nodes and other edges) and constraints. Schemas expressed using other data models may be regarded as representations of schemas of this CDM at a higher level. It is possible to translate any schema expressed using another data model into one expressed using this CDM (see [5]). Thus the local schemas of the local databases, that are expressed in whatever data model their respective databases may be employing, can be automatically translated into component schemas of the CDM.

In this framework these component schemas are integrated into a global schema by means of applying a set of schema transformations to each component schema that transforms it into the global schema. The primitive transformations used add, delete and rename nodes, edges and constraints. A feature of the transformations used is that they are automatically reversible which means that the transformations used to transform a component schema into the global schema may be used to automatically translate queries expressed on the global schema into queries expressed on the component schemas (see [4]). The global queries are written in a common query language (CQL) that uses CDM constructs. Details of the CQL can be found in [8]. The idea is that a global query written in the CQL can be automatically translated, using the schema transformations, into a CQL query using constructs from the component schemas. This query can then be divided into local sub-queries still in CQL that can then be translated into the appropriate query language for each of the local databases and dispatched to them. The global query processor processes the results of these local sub-queries to produce a result for the global query. It should be possible to do all this automatically.

1.3 Aims of the Project

The aim of this project is to develop software that can perform the automatic translation from the original global CQL query to a CQL query expressed using constructs from the component schemas. The software should take, as its input, details of the global schema and component schemas expressed using the CDM, combined with the transformation pathways that transform each component schema into the global schema. Then given a query expressed in the CQL on the global schema it should produce a new query with parts of it now expressed on the component schemas. The project will also look at optimisation of the resulting query.

For the purposes of this project, the constraints part of schemas and schema transformations (see [10]) will be ignored.

2. SPECIFICATION

In this section I discuss the specification of the query translation tool that will be produced by this project.

2.1 Input to the Query Translation Tool

A query expressed on the global schema
This query is written in the CQL (see [8]). Informally the syntax of the CQL is as follows:

```
query   = "[" query "|" qual ";" … ";" qual "]"
        | "[" "]"
        | "group" query
        | "gc" aggFun query
        | query "++" query
        | query "--" query
        | "when" query query query
        | "member" query query
        | "not" query
        | "let" var "=" query "in" query

qual    = pat "<-" query
        | query

pat     = var
        | "(" query "," … "," query ")"

aggFun = "max" | "min" | "count" | "sum" | "avg"
```

Obviously, a more formal syntax will be needed for the actual software. From this informal syntax we can see that a query may be a comprehension (see [8] for discussion of comprehensions), an empty query, the result of various operations applied to other queries such as group, group and compute, append, difference, the conditional operation and so on. A query may also consist of a string, a variable (a 'var' in the above syntax), an expression etc.

Importantly, the CQL supports comprehensions. These have syntax [e|Q_1; … ;Q_n]. Q_1 to Q_n are qualifiers - each is either a filter or a generator. A filter is a boolean valued expression. A generator has syntax p<-s where p is a pattern and s a collection valued expression. Comprehensions can represent directly common operations such as joins, selections and projections. They have the further advantage that anything expressible in the relational algebra is expressible with them.

Some examples of queries might be:

```
Q1) group( _person_qualification )
Q2) [(n)| (c,n)<- course_name; (=)c "CompSci"]
Q3) [(n,p)|(c,n)<-course_name; (c',p)<-course_programme; (=)c c']
```

Example 1 clearly represents a grouping operation. Example 2 represents a selection selecting those course and name pairs where the course is computer science followed by a projection so that the comprehension returns a collection of names only. Example 3 represents a natural join of course_name and course_programme projected to return a collection of name and programme pairs.

The software requires a CQL query expressed on the global schema as input.

<u>The transformations</u>
The transformations from each component schema to the global schema have the following syntax:

```
transformation= “FromSchema” NumToken transf_list “End”

transf_list     = prim_transf SemiColon transf_list
                | prim_transf

prim_transf     = “addNode” nodeSc heme query
                | “delNode” nodeScheme query
                | “addEdge” edgeScheme query
                | “delEdge” edgeScheme query
                | “renameNode” nodeScheme name
                | “renameEdge” edgeScheme name
                | “extendNode” nodeScheme
                | “contractNode” nodeScheme
                | “extendEdge” edgeScheme
                | “contractEdge” edgeScheme

scheme          = nodeScheme | edgeScheme

nodeScheme      = “<<” name “>>”

edgeScheme      = “<<” name “,” scheme_list “>>”

scheme_list     = scheme
                | scheme_list “,” scheme

name            = StrToken
```

From this syntax we can see how schemes are constructed from names, which are merely strings (‘StrToken’s), and other schemes. A schema is merely a set of such node and edge schemes. (A schema also includes constraints which this project will ignore.) Primitive transformations, (the ‘prim_transf’s) take a scheme and possibly another parameter and transform a schema accordingly. The add operations (‘addNode’ and ‘addEdge’) take a scheme and add this scheme to the schema populating it by applying a query (this being their second parameter) to the schema. The delete operations (‘delNode’ and ‘delEdge’) take a scheme and delete this scheme from the schema. They also take a query that gives the extent of the scheme to be deleted. This allows these operations to be automatically reversed if desired. The rename operations behave as you would expect. The extend and contract operations correspond to the add and delete operations respectively but they take only a scheme rather than a scheme and a query. They are equivalent to add and delete operations with empty queries respectively i.e. using our syntax ‘extendNode n’ is equivalent to ‘addNode n []’.

According to the above syntax a transformation from a component schema to the global one starts with the word ‘FromSchema’ followed by a number (a ‘NumToke n’) followed by a list of primitive transformations separated by semicolons followed by the word ‘End’. There follows an example transformation:

```
FromSchema 2
addNode <<“men”>> [p | (p,g) < - person_gender; (=)g “male”];
addNode <<“women”>> [p | (p,g) < - person_gender; (=)g “female”]
End
```

This transformation tells us that to transform component schema 2 into the global schema two things must be done. Firstly, a new node ‘men’ must be added to the schema. This node has the extent given by the query associated with it. In this case the query performs a selection on the pairs in the collection given by person_gender selecting only those whose gender, the second item in the pair, is equal to the string “male”. The result of this selection is then projected o nto the first item in the pair - the person. Thus the new node ‘men’ has the extent of persons whose gender is male. Secondly, a new node ‘female’ must be added to the schema. The extent of this is determined in an analogous manner to new node ‘male’.

The software will require transformations expressed using the above syntax as input.

<u>The global schema definition</u>
The syntax for schemes was given in the previous section as was the syntax for a ‘scheme_list’. The software will require as input the defin ition of the global schema given in the form of a ‘scheme_list’. For example, the following ‘scheme_list’ defines a global schema, S, consisting of four entities (person, address, qualification, post) and three associations between person and address, qualification and post respectively:

```
<<"person">>,
<<"address">>,
<<"qualification">>,
<<"post">>,
<<"",<<"person">>,<<"address">> >>,
<<"",<<"person">>,<<"qualification">> >>,
<<"",<<"person">>,<<"post">> >>
```

<u>The simplification rules</u>
The purpose of simplifications is to optimise the query that results from the translation for faster processing on the local databases. This is accomplished by the application of various simplification rules to the query.

The syntax for a simplification rule is as follows:

simplification = “From” query “To” query

Simply the word ‘From’ followed by a query then the word ‘To’ followed by a query. These will not be ordinary queries however. Indeed, they will not actually be queries at all in a semantic sense. They will be queries according to the syntax of the CQL but will contain within them special variables that do not correspond to anything in any of the schemas but instead serve as placeholders for anything that might appear at that point of the query. Thus these special queries will in fact be query templates giving a general form a query might take. The simplification thus communicates that any query of a certain form can and should be replaced by a query of another form that is equivalent to it. The syntax for expressing simplification rules and the mechanism for implementing them is entirely my own work and therefore there is no need to say at the specification stage what special variables the query templates will contain and how they will be recognised. However, examples of simplification rules might be:

1) From [] ++ xxx1 To xxx1
2) From group(xxx1 ++ xxx2) To group(xxx1) merge group(xxx2)

Example 1 states that a query having the form of an empty query and something else (syntactically it must be another query) appended to it can be simplified to just the something else. Example 2 states that if a query consists of one query, $q_{1,}$ appended to another, q_2, and a group performed on the result then this may be simplified (in some sense) to consist of the result of a merge of g_1 and g_2 where g_1 is the result of a group applied to q_1 and g_2 is the result of a group applied to q_2. In these examples xxx1 and xxx2 are the special variables.

The software will require as input a set of these simplification rules.

2.2 General Issues

The implementation language will be Java. For the purposes of the project constraints will be ignored. The software will assume all input to be error free and will therefore not incorporate any error handling.

We also need some convention to translate schemes in transformation lists and schema definitions into identifiers in CQL. Let G denote this translation function for any node scheme <<“n”>> or edge scheme <<“n”, s1, ..., sn>> in the global schema. Then

G[<<“n”>>] = “n”
G[<<“n”, s1, ..., sn> > = “n” ++ “_” ++ G[s1] ++ “_” ++ ... ++ “_” ++ G[sn]

Similarly let Li denote this translation function for scheme in component schema i. Then

Li[s] = G[s] ++ “i”

These translation functions G and Li were specified by my supervisor.

2.3 Output from the Query Translation Tool

The output of the software will be a rewritten query expressed in the CQL. All global schema constructs in the original query will have been replaced by sub-queries using component schema constructs. The simplification rules will also have been applied.

Here is an example that illustrates the whole translation process. Consider the example query Q1, group(_person_qualification) on the global schema, S, given as an example earlier. Suppose that S has been derived from three component schemas by the following three lists of transformations:

```
FromSchema 1
addNode <<"person">> men1 ++ women1;
addEdge <<"", <<"person">>, <<"post">> >> _men_post1 ++_women_post1;
contractEdge <<"", <<"men">>, <<"post">> >>;
contractEdge <<"", <<"women">>, <<"post">> >>;
contractNode <<"men">>;
contractNode <<"women">>;
extendNode <<"address">>;
extendEdge <<"",<<"person">>,<<"address">> >>;
extendNode <<"qualification">>;
extendEdge <<"",<<"person">>,<<"qualification">> >>
End

FromSchema 2
renameNode <<"name">> "person";
extendNode <<"post">>;
extendEdge <<"",<<"person">>,<<"post">> >>
End

FromSchema 3
renameEdge <<"quals", <<"person">>, <<"qualification">> >> "";
extendNode <<"address">>;
extendEdge <<"", <<"person">>, <<"address">> >>
End
```

Then the query Q1 should translate into the following query (note that identifiers from schema i are suffixed with the number i):

```
group ( [] ++ _name_qualification2 ++ quals_person_qualification3 )
```

We would then want the resulting query to be simplified using simplification rules (the ones given earlier will be sufficient if applied correctly) to give something like:

```
group( _name_qualification2 ) merge group ( quals_person_qualification3 )
```

3. DESIGN

The input to the program is in the form of text. The output will also be text. Thus, the input will have to be separated into its component symbols by a lexer then converted into some abstract form, a tree, by the parser prior to its being processed. The processing will consist of making appropriate changes to this abstract representation to accomplish the translation and simplification according to the rules. Then to output the processed query its tree representation will have to be turned back into text prior to output. This quick sketch of a design leads into more detailed design issues.

3.1 Design of lexer and parser

The need for a lexer and a parser are obvious. The text input must be turned into some abstract internal representation for processing. Rather than code the lexer and the parser from scratch in Java I decided to use existing tools for the job. The reasons for this decision are obvious. Performance is not a primary issue for this project, what is required is that the lexer and parser work correctly. Under these circumstances to code from scratch would just be wasting time. It was decided to use a tool called JLex (see [1]) to write the lexer and a tool called CUP (see [2]) to write the parser these being recommended by my supervisor and being designed to work together for use with and written in Java.

There will be just one lexer and just one parser. These will deal with all the different types of input. Although the input text file may contain a query or a list of simplification rules or a list of transformations I feel it is appropriate to have just one lexer and one parser to deal with all the various varieties of input. This is because transformations may contain queries and simplification rules always do. To use different lexers and parsers for different types of input would lead to either unnecessary complexity or duplication of code.

The lexer will behave in the obvious manner. It will take a text file as input and as output will return, one by one as requested, the symbols that the text in the text file represents. The JLex tool takes a text file containing instructions detailing how the lexer is to work and from this produces a Java file that implements the lexer. Thus, there will be a class for the lexer.

The parser takes as its input the symbols passed, one by one, from the lexer. From these it constructs a tree in memory. This is the output it returns. The tree may represent a query, a list of simplification rules or a list of transformations. If it represents transformations it will also represent the global schema - these both concern the schema transformation process and will come from the same text file. Like JLex, CUP takes a text file telling it how to build the parser as input and produces Java files that compile to classes to implement it. Thus, there will be a parser class in the software.

3.2 The Structure of the Abstract Syntax Tree

The same sort of tree will be used for representing queries, transformations and so on. The reasons for this are those outlined above for having one lexer and one parser for all input. However, other data structures will be used to represent the things other than queries when they are actually needed for processing. It is natural though for them to be initially represented as trees when first they are read in from text files.

The trees will be binary trees. It may seem natural for the nodes of the tree to be allowed more than two children. Take, for example, a comprehension with the structure [e | Q_1; ... ; Q_n]. It may seem natural for this to be represented by a comprehension or a query node with n+1 children, the first child representing e, the second Q_1 and so on. If the parent node in this structure contained information that it was representing a comprehension this structure might seem the most obvious and easiest way to represent a comprehension. It is certainly unambiguous. However, I decided to use binary trees because I thought they would be more straightforward to traverse. It also occurred to me that being of a more regular structure it might be easier to create query templates using them and thus make processing the simplifications easier. Binary trees will be deeper but this doesn't really matter.

I decided that the tree should retain as much information from the text representation as possible. I will explain what I mean by an example. Take the form of a simplification rule. The text form is the word 'From' followed by a query followed by the word 'To' followed by another query. Without ambiguity the tree representation of this could consist of node to represent the simplification having two children the first representing the first query and the second the second. In fact the tree representation will consist of a node representing the simplification having among its descendents nodes representing not only the two queries but also the words 'From' and 'To'. The purpose of this redundancy is to make reconstructing the original text easier. To reconstruct the original text without this redundancy would require adding in extra bits of text implicit in the tree as the tree was traversed. The software would need to know that a simplification rule required the words 'From' and 'To' adding back in. With the redundancy, the tree will be so constructed that a one simple traversal will be all that is required to reconstruct the original text. While this is not particularly useful for simplification rules it is very useful for queries. After a query has been processed it must be converted back to text. To be able to do this easily is very useful and justifies a certain amount of redundancy in the information stored in the tree.

The tree will obviously be made up of node objects. The code will thus require a node class. What functionality should this node class have? Each node will need to store information regarding what structure its sub-tree represents (if it represents a valid structure in its own right), what text is associated with the node (i.e. the text, if any, for the textual representation of the structure) and links to its left and right child. Because nodes may often have to be treated as the root of their sub-trees representing their sub-structures, the node class may require some extra functionality on top of this simple design.

Figure 1 at the end of this section shows a tree representing the query 'group (_person_qualification)'. It also shows that even for a very simple query the tree is quite large.

3.3 Query Translation

In broad terms the job of the actual translation software is to take trees representing a query, the global schema and transformation rules and to produce a tree representing the query translated with reference to the global schema and transformation rules. One possible design for this software would involve taking the query tree, traversing it to find references to global schema constructs and then using the transformation rules to replace these global schema constructs with appropriate queries on the component schemas. This is not the design I decided to use. It makes more sense to initialise the translation software once with the global schema and transformation rules and have it store mappings from global schema constructs and queries expressed on the component schemas. Then when passed a query tree to translate it can traverse this tree and replace each global schema construct it finds with the appropriate query using the mappings it has stored. This is clearly a more efficient way of accomplishing the translation.

This translation will be accomplished using a substituter class (so called because it performs substitutions in the query tree). Its constructor will take a tree, generated from a transformations file, representing both a global schema and a list of transformations from component schemas. It will generate the mappings from this tree and store them as part of the data for substituter objects. Substituter objects will have a method that will take a query and perform the appropriate substitutions in it to translate it onto the component schemas.

Once the mappings are generated the actual process of performing substitutions will be fairly straightforward. Generating the mappings from the tree representing the global schema and the transformation list requires more thought however. Firstly, a list of the global identifiers that may appear in global queries will be generated from the global schema. Coding this should be reasonably simple. The sub-tree representing the global schema can be traversed to find each scheme sub-structure. A function to convert a scheme sub-tree into a string identifier will then be used and each identifier added to a list stored by the substituter object. When this list is generated it can be used to help generate the mappings.

If a global construct is not mentioned in a transformation list from a component schema to the global schema this means that no transformation is performed on it. Using the example given during the specification, the transformation from component schema 3 to the global schema does not do anything to the 'person' node. Thus identifie r 'person' in the global schema maps to identifier 'person3' in component schema 3. To generate all the mapping for component schema 3 we may initially assume that 'person' maps to 'person3', 'address' maps to 'address3' and so on. We then process its pr imitive transformations in reverse order changing the mappings as we go. So firstly we would make '_person_address' map to the empty query '[]'. Then we would make 'address' map to the empty query '[]'. Finally, we would make '_person_qualification' map to 'quals_person_qualification'. We could perform this process for each transformation and end up with a set of mapping from global identifiers for each component schema. Following the running example, 'person' maps to 'men1 ++ women1' for component schema 1, 'person' maps to 'name2' for component schema 2 and 'person' maps to 'person3' for component schema 3. We can then put all these mappings together to get a general mapping from global identifiers to queries expressed using component schema constructs. So 'person' would map to 'men1 ++ women1 ++ name2 ++ person3'. This is, in essence, how the mapping will be generated although in practise rather than store a set of mapping for each component schema it will be easier to generate the mappings for schema 1, add these to the general mappings, generate the mappings for schema 2, add these to the general mappings and so on and so on.

As an aside, it should be noted that as far as translation from global queries goes, which is after all what this project is concerned with, schemes deleted or contracted from component schemas are irrelevant – they have no bearing on translation from global queries because they are not present in the global schema to be translated. I will include them in my examples, however, for completeness and the software should be able to cope with their presence even if by coping with them I only mean ignoring them.

So, to summarise, the constructor for the substituter object will be passed a tree representing both the global schema and the transformations from the component schemas. From this it will generate, using the method outlined above, a list of global identifiers and a set of mappings from global identifiers to queries expressed in terms of the component schemas. These will be stored as part of the data of the object. Substituter objects will have a method that when passed a query tree will use these mappings to substitute for the global identifiers in the query tree the appropriate component queries. Thus, query translation from global queries to queries expressed using component schema constructs will be achieved.

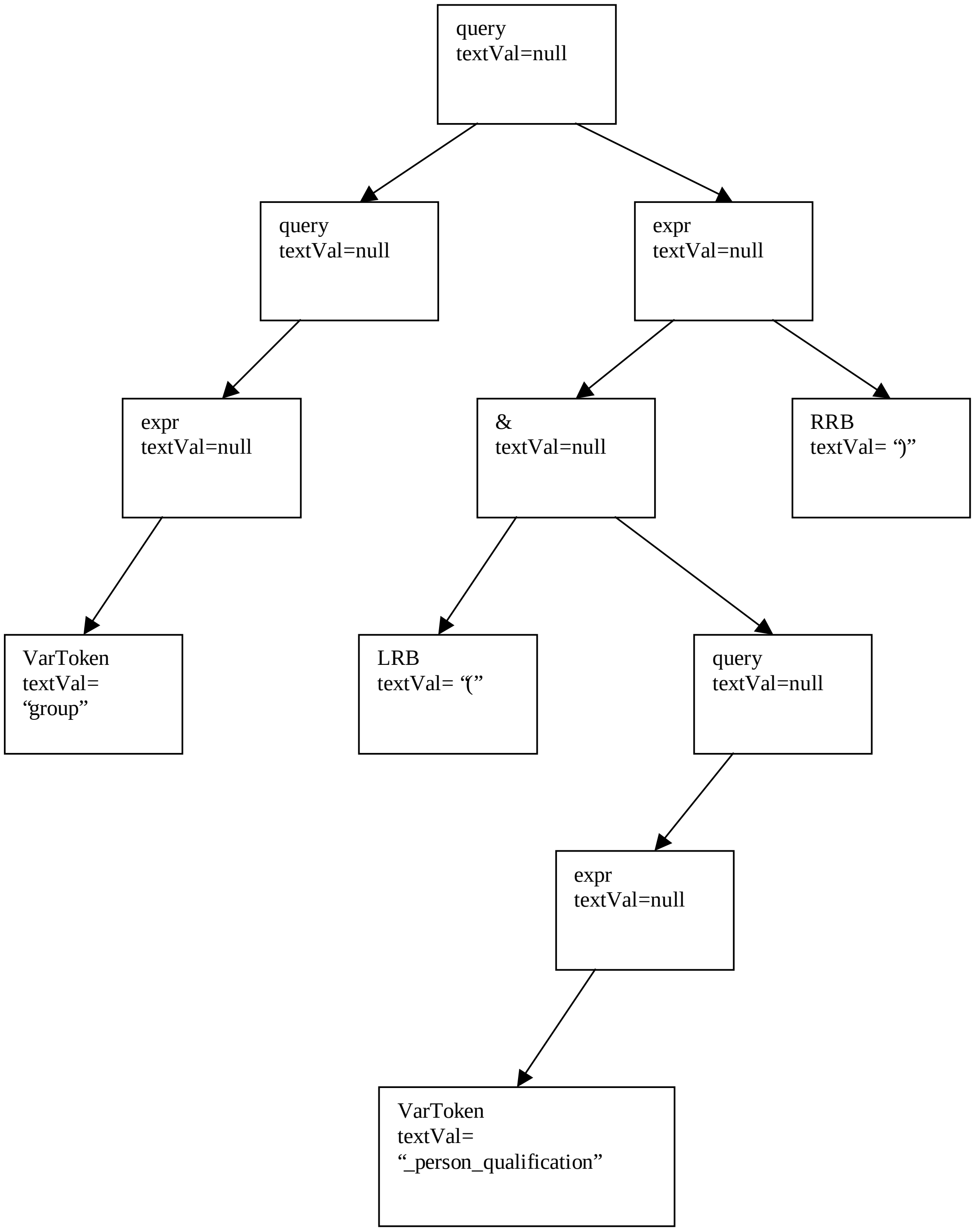
3.4 Query Simplification

The lexer and parser will convert the text file containing the simplification rules into a tree as explained above. It will then be necessary to convert this tree into some data structure representing 'from' and 'to' query templates for use when simplifying queries. The query templates themselves will remain in the form of trees. A list structure will store pairs of these templates representing from-to simplifications. The actual process of simplification will proceed as follows. The query to be simplified is in the form of a tree. This query tree will be traversed in-order. Each node will then be checked to see it its sub-structure corresponds to one of the 'from' query templates in the list - going through the list in order. If it does correspond its sub-tree will be changed to have the same shape as the appropriate 'to' query template in the list of pairs. This process will be repeated until a traversal of the query tree is completed with no alterations, indicating that all simplifications have been performed. The process needs to be repeated as a simplification of a particular structure in a query might cause a structure of which the simplified structure is a sub-structure to require simplifying. Only a complete tree traversal without change can make sure that all simplifications that can be done have been done.

This simple algorithm for performing the simplifications has various inadequacies. Firstly, it is probably not very efficient. Secondly, the order in which simplifications are performed - which may affect the final query produced - is rather arbitrary being as it is determined more by convenience than logic. However, this algorithm should work so is fine as a first attempt. Furthermore, by using multiple simplification list text files and performing this whole process multiple times it is possible to exercise more control over the order of simplifications if desired.

To accomplish all this there will be a simplifier class. The constructor for this class will take a tree representing a list of simplification rules. It will traverse this tree to generate the list of pairs of templates that will form part of the data for the simplifier object. Simplifier objects will have a method that will take a query and perform the appropriate simplifications on it.

Figure1. The abstract syntax tree for the query ‘group (_person_qualification)’. Each node contains information about what structure its sub-tree represents (query, expression etc.) and also an associated text value.



4. IMPLEMENTATION

4.1 Research and Reading

The subject matter of this project goes considerably beyond the material covered in the Databases lectures on the MSc course. This meant that a large amount of preparatory reading and research was required before the design of the software could even begin. It was necessary to learn a relatively substantial volume of material connecting to the framework within which I was to work. I had to learn about the hypergraph data model. I had to learn how the automatically reversible schema transformations worked. ([3], [4], [5], [6], [7] and [10] were all relevant to these goals). It was also necessary for me to understand the workings of the common query language (CQL) to be used (I used [8]). Obviously, as well as reading I needed to spend a reasonable amount of time actually working through the mechanics of the framework on paper as well. I spent approximately a month and half on the above. As well as this, during this time, my supervisor explained to me the precise syntax of the query language and the transformation language. We also discussed various things that could be attempted beyond the basic automatic query translation. In the end, of these extensions of the project, there was only time to actually attempt to implement the simplification of translated queries.

4.2 Parsing the Input

The first thing to implement was software to parse input into trees so that it could be processed. My supervisor recommended the use of the tools CUP and JLex for this. Having never used these tools before – indeed, having no previous experience with parsing – it was necessary to dedicate some time to learning how to use them. For this I read the manuals downloaded with the software (see [1] and [2]). After about two weeks I felt confident enough to move onto coding.

The file 'EdLex1' is the file passed to JLex for it to create the lexer from. Basically, it lists what symbols are to be looked for in the text file to be analysed and gives code to be executed when these symbols are found. 'EdLex1' also contains code for a 'main()' method. This is not to be used by my finished software but can be used to test that the lexer is producing the expected results by outputting information about each symbol the lexer produces. From 'EdLex1' JLex generates a Java source code file 'EdLex1.java'. When compiled two class files are produced 'EdLex1.class' and 'Yylex.class'. The EdLex1 class can be used for testing as explained above. The Yylex class is what is actually used for lexing by the other software. Yylex objects are created with an associated source of input – text files for the purposes of this project – and contain a next_token() method that, unsurprisingly, returns the next token.

The file 'EdCup1.cup' is the file passed to CUP that it uses to create the actual parser. This file starts by listing the various symbols that the text to be parsed may contain including non-terminal symbols. A syntax is then specified with code included that gives instructions on how to construct each of the sub-structures that might make up the resulting tree. From 'EdCup1.cup' CUP generates two Java source code files, 'sym.java' and 'parser.java' which compile to 'sym' and 'parser' classes. The sym class contains only static final int variables – one for each terminal symbol. These are needed for use by the lexer. I also chose to use them elsewhere in the software. For this purpose I created another class, Elements, containing static final int variables for each non-terminal symbol. The parser class is what actually does the parsing. When a parser object is created the constructor is passed a Yylex object (which has a source of input associated with it). Thus, the parser receives tokens from the lexer. The parser object has a method parse() which parses all tokens it receives from the Yylex object into a tree and returns an object that contains a reference to the root of the tree.

Trees are made up of nodes. Thus, as I was writing the lexer and the parser I also needed to write a node class. The code for this is in file 'Node.java'. As discussed earlier, this needs member variables to store references to the left and right children, to a string containing the text for the node and to store an int giving which symbol, terminal or non-terminal, the node and its sub-tree represent. At this stage, I included all these variables and also various constructors to be used by the lexer and the parser. The methods for this class were not needed at this stage and were not written until later.

When the above was written and working the software was capable of parsing input from text files into trees. Combined with other code that was evolved during the project, now surviving as the printQuery() and outputTree() methods of the TestDriver class, it was possible to check that everything was working correctly. This work on producing a functional parser took approximately one month.

4.3 Translating the Query

The next thing to implement was software for translating queries based on transformations. This is accomplished using the Substituter class the source code for which is in file 'Substituter.java'. This class has one constructor and one public method, performSubs().

The constructor is passed a Node reference which is the root of a tree generated by parsing a transformations text file. Its purpose is to initialise an ArrayList object, globalnames, which contains a list of the names of the constructs of the global schema and to initialise a HashMap object, mainMap, which contains mappings from names of global constructs to queries expressed using component schema constructs. The globalnames object is initialised fairly easily. The recursive method buildGlobalNames() is called which traverses the appropriate section of the input tree finding schemes for the global constructs, converting them into identifiers, and adding these identifiers to globalnames. This method makes use of the scheme_to_name() method which in turn calls the recursive s_to_n() method which finds the 'StrToken's in the scheme and concatenates them together in the appropriate manner to give the name of the identifier for the scheme.

Initialising the mainMap object is more complicated. Firstly, it is set up so that every global name maps to an empty query (making use of the static final emptyQuery Node reference). Then buildSubMap() is called, being passed a reference to the root of the part of the input tree containing the transformations. The buildSubMap() is recursive and traverses the tree looking for transformations from component schemas to the global schema. Whenever it finds a transformation is creates a HashMap(), tempmap, which is initialised to map each global construct to itself (i.e. it assumes no changes are made to it initially). This tempmap object is then passed to the process_tran() method along with the Node reference that is the root of the transformation. The process_tran() method returns having altered tempmap so that now it maps each global name to the appropriate query for the component schema the transformation relates to. Using tempmap, mainMap is then updated so that the query for every global name in tempmap is appended to the query for the same global name in mainMap. Obviously, when buildSubMap() is finished mainMap contains mappings from names of global identifiers to queries expressed on the component schemas that give the correct result.

The process_tran() method works as follows. It is passed a reference to the tempmap HashMap as explained above. It is also passed a Node reference to the transformation it is meant to process. It is a recursive method, which goes through the primitive transformations that make up the transformation it is to process, in reverse order. It must go through them in reverse order for the following reasons. Each transformation consists of a series of primitive transformations that transform a component scheme into the global schema. Each primitive transformation is automatically reversible. Thus, reversing first the last primitive transformation done, then the second last and so on can reverse the whole transformation. Thus process_tran() processes the primitive transactions in reverse order making the appropriate substitutions in the queries in tempmap.

From the above we can see how the constructor initialises the mainMap and globalnames objects. We can now turn to the Substituter class's only public method, performSubs(). The workings of this method are simple. It is a recursive method that is passed a Node reference. The sub-tree of this node is traversed recursively and instances of global identifiers are found they are replaced in the tree by queries expressed in terms of the component schemas. The mainMap HashMap is used.

To accomplish all the above extra functionality had to be added to the Node class. The copySubtree() and copyNode() methods were added. The Substituter class uses these methods. The copySubtree() method returns a node that is the root of a tree that is a copy of the sub-tree of the node whose copySubtree() is called. The copyNode() method takes a Node reference as a parameter. Calling a Node object's copyNode() method causes that node to become the root of a tree that is a copy of the tree of the parameter Node object.

Substituter objects enable queries to be translated using their performSubs() methods. This section of the implementation took approximately two weeks. Again more code was needed to test that is was working and some of this survives in the TestDriver class.

4.4 Simplifying the Query

Next, I needed to write the software for simplifying the queries once translated. This job is done by the Simplifier class the source code for which is in file 'Simplifier.java'. This class has one constructor and one public method, simplify().

The constructor is passed a Node reference. This node is the root of a tree resulting from parsing a simplifications text file. Simplifier objects have a member variable, querypairs, and the purpose of the constructor is to initialise this. The querypairs object is an ArrayList whose function is to hold pairs of query templates – these being the 'from' and 'to' query template pairs t o perform simplifications. It is initialised in the following way. The constructor calls a recursive method, setItUp(), that traverses the input tree. For each simplification structure it finds in the tree it adds a query pair to querypairs. These query pairs are the 'from' and 'to' templates for the simplification. The constructor then calls the cleanTemplates() method. This method alters the query templates in querypairs slightly to make the process of comparison with real queries slightly easier.

Simplifier's public method, simplify(), is used to perform the actual simplifications. It is passed a Node reference. This node is root of the tree representing the query to be simplified. The simplify() method calls the simp() method which traverses the tree performing simplifications and returns a value of true if any simplifications were needed. It repeatedly calls simp() until such time as simp() returns a value of false indicating no simplifications were performed and thus no more will be necessary. The simp() method itself is recursive, it calls check() on every node in the tree. Thus check() is performed on every sub-tree in the query tree. The check() method itself calls the compare() method once for every pair of templates in querypairs. The compare() method calls the comp() method passing it the root node of the sub-tree to be checked, the root node of the 'from' template of the pair being processed and an empty HashMap, mapvar. The comp() method is a recursive method that checks whether a sub-tree and a query template have the same structure – a structure requiring simplification. It returns true if this is the case. It also finds placeholder identifiers in the template and adds mappings to the mapvar object mapping from each placeholder identifier in the template to the sub-query to be substituted for the identifier in the 'to' template. The compare() method then, provided comp() returned true, changes the sub-tree to simplify its structure using the 'to' template from the pair and usi ng mapvar.

Thus, queries may be simplified using the simplify() methods of Simplifier objects. This section of the implementation took approximately two weeks.

<u>4.5 Testing the Software</u>

By the time I had completed the above sections of implementation there was very little time left for testing. However, some rudimentary testing was done. As part of the testing process I needed to complete the TestDriver class. In its final version, it has a main() method that displays a menu offering options of loading a transformations file, displaying the substitutions made during translations, loading a query, outputting the query, outputting a text representation of the structure of the query tree, translating the query, loading a simplifications file and performing simplifications. I used this class to help test the rest of the software. (The code to implement the menu in the TestDriver class was adapted from code written by Keith Mannock of Birkbeck College. It uses the TextIO class written by David Eck of Hobart and William Smith College, Geneva. I could easily have written the menu from scratch but did not do so in an effort to save time.)

As far as I could ascertain the query translation part of the software worked as would be desired. I will use the running example from Chapter 2 of this report. Given the global schema, S, and transformation list, stored in the file 'TFile1.txt' this was successfully parsed and the following output was given when the test driver was asked to display the substitutions to be made during translations:

```
person
[ ] ++ ( men1 ++ women1 ) ++ name2 ++ person3
address
[ ] ++ ( [ ] ) ++ address2 ++ ( [ ] )
qualification
[ ] ++ ( [ ] ) ++ qualification2 ++ qualification3
post
[ ] ++ post1 ++ ( [ ] ) ++ post3
_person_address
[ ] ++ ( [ ] ) ++ _name_address2 ++ ( [ ] )
_person_qualification
[ ] ++ ( [ ] ) ++ _name_qualification2 ++ quals_person_qualification3
_person_post
[ ] ++ ( _men_post1 ++ _women_post1 ) ++ ( [ ] ) ++ _person_post3
```

This output indicates that the Substituter object has 'person' mapping to '[] ++ (men1 ++ women1) ++ name2 ++ person3' and so on. The Substituter object has global identifiers mapping to queries expressed on component schemas and these appear to be accurate. When asked to translate the example query 'group (_person_qualification)' and then output the result it gave:

```
group ( ( [ ] ++ ( [ ] ) ++ _name_qualification2 ++ quals_person_qualification3 ) )
```

This is a query that would give the correct results. I tentatively conclude that the query translation software works but clearly more substantive testing remains to be done.

This example reveals certain problems with the simplification software, however. Consider the following simplification rules contained in file 'SFile1.txt':

```
From group(xxx1 ++ xxx2) To (group(xxx1)) merge (group(xxx2))
From xxx1 ++ [] To xxx1
From [] ++ xxx1 To xxx1
From ([]) To []
From ((xxx1)) To (xxx1)
From xxx1 ((xxx2)) To xxx1 (xxx2)
```

There are sufficient rules here to simplify the translated query given above. When these rules are given to a Simplifier object, however, and simplifications performed the result is the following:

```
( ( group ( [ ] ) ) merge ( group ( _name_qualification2 ) ) ) merge ( group (
quals_person_qualification3 ) )
```

This is not the desired result. The problem clearly arises from the lack of control over the order in which rules are performed. The '([])' is separated out to have its own group operation before it is got rid of by the other rules. It thus escapes being got rid of. The problem for this query could be solved by adding extra simplification rules but I didn't do this because I wanted to illustrate the potential problems of lacking control over the order the simplification rules are performed in. To further illustrate this point consider the following. 'SFile2.txt' consists of:

```
From xxx1 ++ [] To xxx1
From [] ++ xxx1 To xxx1
From ([]) To []
From ((xxx1)) To (xxx1)
From xxx1 ((xxx2)) To xxx1 (xxx2)
```

and 'SFile3.txt' of

```
From group(xxx1 ++ xxx2) To (group(xxx1)) merge (group(xxx2))
```

If, using the test driver, we translate the query as above then load 'SFile2.txt' and perform simplifications we get:

```
group ( _name_qualification2 ++ quals_person_qualification3 )
```

If we then load 'SFile3.txt' and perform simplifications we get:

```
( group ( _name_qualification2 ) ) merge ( group ( quals_person_qualification3 ) )
```

This is essentially what is desired. Using an, admittedly ad hoc, method to control the order of simplifications we can get the desired result. There were other problems with the simplification software but these were problems I expected all along, so were not discovered as a result of testing so are not discussed here but later.

Other testing was performed but it was all of the same nature as that given above. I did not perform anything that could be considered truly rigorous systematic testing. Firstly, I did not have time. Secondly, I have insufficient knowledge of how this software will need to perform in practice, what might be considered typical queries and typical transformations, to test for these things. The testing was therefore limited to things like that given above. However, even in the example given, the software must produce mappings from global identifiers using transformations that include adding, extending and renaming both nodes and edges. So a certain range of things are tested.

To conclude, the testing indicated that the software essentially worked as planned with some limitations on the simplifications. However, more rigorous testing is clearly needed. Finishing the TestDriver class, testing, debugging and some writing up were done in parallel for about two weeks. Most of this time was spent on debugging.

5. CONCLUSIONS

5.1 Problems with the Project

The main problem with the finished software lies with the simplification of the translated queries. As explained in section 4 this does not work as intended. The user does not have proper control over the order in which simplifications are performed and this can lead to undesired results. This problem can be avoided by using multiple Simplifier objects but this is not an ideal solution. It renders control of the order in which simplifications are performed something that has to be done in code. It would be much better if this control were available by instructions contained in the simplifications text file. A further problem with the query simplification part of the software comes from the way I designed the syntax of simplifications. The 'from' and 'to' query templates must syntactically be queries. If they are not the parser will produce an error. This limits the type of simplifications that can be performed and prevents certain desirable ones from being implemented. For example, it is not possible to specify simplifications for lists of qualifiers within comprehensions. A list of qualifiers is not a query so this cannot be done. It may, however, be desirable that it is done.

Apart from the above problems the actual software works as intended as far as my testing has been able to ascertain. The are, however, further problems with the project as a whole. Due to poor time management a disproportionate amount of time was spent on certain activities. Had it not been for this better code may have been produced. I found the code I had written difficult to debug when it came to this stage and by this stage there was insufficient time to drastically change it to make it easier to debug. The biggest problem caused by not having sufficient time was the lack of testing. I have confidence in the design of the software, with the exception of those deficiencies already mentioned, but I cannot state with certainty that the actual code written will always work. This is because of inadequate testing.

5.2 Possible Improvements to the Software

There are a number of possible improvements that could be made to the software. Obviously, the simplification of queries could be improved as discussed above. Another possible improvement, would be the extension of the transformation language. An enhanced transformation language is outlined in [9]. Transformations in the existing language are schema specific in that they are meant only to transform a single schema, A, into a single schema, B. In the extended language, parameterised transformations can be written. Thus, one parameterised transformation could take any schema, Ai, from a set of schemas, A1, A2 ... An, and transform it into a schema, Bi - its equivalent schema from B1, B2 ... Bn. This would obviously constitute a major extension to the software as it currently is.

Another addition to the software would be to enable it to translate the Common Query Language from and to SQL. It is likely that a user would want to query the heterogeneous database using SQL, or another high-level query language, rather than the CQL. It is also likely that the local databases would need to be queried using SQL. Translating to and from SQL, and probably other query languages, would be a necessary addition to make a practically useful system.

5.3 Final Conclusions

There were various problems with some aspects of the final software. Problems with time management during the project led to inadequate testing. Despite this the central goal of the project was to produce software to automatically translate queries expressed on a global schema into ones expressed on various component schema using the transformation pathways in between and this central goal was achieved. Even if the software does turn out to have bugs my testing did not detect I believe the basic design to be sound. The software I have written and work I have done provide a good basis for the extensions outlined above.

REFERENCES

[1] Berk E, "JLex: A lexical analyzer generator for Java", http://www.cs.princeton.edu/~appel/modern/java/JLex, 1997.

[2] Hudson S, "CUP User's Manual", http://www.cs.princeton.edu/~appel/modern/java/CUP, 1999.

[3] McBrien P and Poulovassilis A, "A formal framework for ER schema transformation", Proc. ER'97, volume 1331 of LNCS, pages 408 -421. Springer-Verlag, 1997.

[4] McBrien P and Poulovassilis A, "Automatic migration and wrapping of databases applications – a schema transformation approach", Proc. ER'99, volume 1728 of LNCS, pages 96-113. Springer-Verlag, 1999.

[5] McBrien P and Poulovassilis A, "A uniform approach to inter -model transformations", Advanced Information Systems Engineering, 11th International Conference CAiSE'99, vo lume 1626, pages 333-348. Springer-Verlag, 1999.

[6] McBrien P and Poulovassilis A, "Schema evolution in heterogeneous database applications, a schema transformation approach". Technical Report 31/03/00, Birkbeck College and Imperial College.

[7] Poulovassilis A, Lecture course entitled "Advances in Databases" (Notes 4 – Distributed Databases, Notes 5 – Heterogeneous Databases). July 2001.

[8] Poulovassilis A, "Automed working document 2, the Automed Intermediate Query Language". Technical Report 15/06 /01, Birkbeck College.

[9] Poulovassilis A, "Automed working document 4, An enhanced transformation language for the HDM". Technical Report 31/06/01, Birkbeck College.

[10] Poulovassilis A and McBrien P, "A general formal framework for schema transformation", Data and Knowledge Engineering, 28(1), pages 47 -71, 1998.

APPENDIX A – Java Files

Node.java

```
package QTran;

import java.io.*;
import java_cup.runtime.*;

public class Node
{
  int snum;
  Node lRef, rRef;
  String textVal;

  public Node(int snum, Object lRef, Object rRef)
  {
    this.snum = snum;
    this.lRef = (Node)lRef;
    this.rRef = (Node)rRef;
  }

  public Node(int snum, String textVal)
  {
    this(snum, null, null);
    this.textVal = new String(textVal);
  }

  public Node(Object lRef, Object rRef)
  {
    this(Elements.PARTIAL_STRUCTURE,lRef,rRef);
  }

  public Node(InputStream inpstr) throws Exception
  {
    parser p = new parser(new Yylex(inpstr) );
    Symbol s = p.parse();
    Node n = (Node)s.value;
    this.snum = n.snum;
    this.lRef = n.lRef;
    this.rRef = n.rRef;
    this.textVal = n.textVal;
  }

  private Node()
  {
  }

  public void copyNode(Node n)
  {
    if (n==null) throw new NullPointerException();

    snum = n.snum;

    if (n.textVal!=null)
      textVal = new String(n.textVal);
    else textVal = null;

    if (n.lRef!=null)
      lRef = n.lRef.copySubtree();
    else lRef=null;
    if (n.rRef!=null)
      rRef = n.rRef.copySubtree();
    else rRef=null;
  }

  public Node copySubtree()
  {
    Node n = new Node();
    n.snum = snum;

    if (textVal!=null)
      n.textVal = new String(textVal);
    else n.textVal = null;

    if (lRef!=null)
      n.lRef = lRef.copySubtree();

    if (rRef!=null)
      n.rRef = rRef.copySubtree();

    return n;
  }

}
```

Substituter.java

```
package QTran;

import java.util.*;

public class Substituter
{
  static final Node emptyQuery =
    new Node(Elements.query,
      new Node(Elements.expr, new Node(sym.LSB, "["), new
Node(sym.RSB, "]") ), null);

  HashMap mainMap;

  ArrayList globalnames;

  private String cleanStr(String s)
  {
    return s.substring(1, s.length() - 1);
  }

  private void s_to_n(StringBuffer s, Node root)
  {
    if (root==null) return;
    if (root.snum==sym.StrToken)
    {
      s.append(cleanStr(root.textVal));
      s.append("_");
      return;
    }
    s_to_n(s, root.lRef);
    s_to_n(s, root.rRef);
  }

  private String scheme_to_name(Node root)
  {
    StringBuffer s = new StringBuffer();
    s_to_n(s, root);
    s.deleteCharAt(s.length()-1);
    return s.toString();
  }

  private void buildGlobalNames(Node root)
  {
    if (root==null) return;
    if (root.snum == Elements.scheme)
    {
      globalnames.add(scheme_to_name(root));
      return;
    }
    buildGlobalNames(root.lRef);
    buildGlobalNames(root.rRef);
  }


  private void update_tree(Node n, String s, Node q)
  {
    if (n==null) return;
    if (n.snum == Elements.expr)
      if (n.lRef.snum == sym.VarToken)
```

```
        if (n.lRef.textVal.equals(s))
        {
          n.copyNode( new Node(Elements.expr, new Node( new
Node(sym.LRB, "("), q),
            new Node(sym.RRB, ")") ) );
          return;
        }
    update_tree(n.lRef, s, q);
    update_tree(n.rRef, s, q);
  }

  private void update_map(HashMap tempmap, String s, Node q)
  {
    for (int i=0; i<globalnames.size(); i++)
    {
      Node n =
((Node)(tempmap.get(globalnames.get(i)))).copySubtree();
      update_tree(n,s,q);
      tempmap.put(globalnames.get(i), n.copySubtree() );
    }
  }

  private void text_rep(Node n, String f, String t)
  {
    if (n==null) return;
    if (n.snum==sym.VarToken)
    {

      StringBuffer sb = new StringBuffer();
      int i = 0;
      int ni = n.textVal.indexOf(f);
      while (ni!=-1)
      {
        sb.append( n.textVal.substring(i,ni) );
        sb.append( t );
        i = ni + f.length();
        ni = n.textVal.indexOf(f,i);
      }
      sb.append( n.textVal.substring(i) );
      n.textVal = sb.toString();
      return;
    }
    text_rep(n.lRef,f,t);
    text_rep(n.rRef,f,t);
  }

  private void text_replace(HashMap tempmap, String f, String t)
  {
    for (int i=0; i<globalnames.size(); i++)
    {
      Node n =
((Node)(tempmap.get(globalnames.get(i)))).copySubtree();
      text_rep(n,f,t);
      tempmap.put(globalnames.get(i), n.copySubtree() );
    }
  }

  private void process_tran(Node root, HashMap tempmap, String num)
  {
    if (root==null) return;
    if (root.snum == Elements.prim_transf)
    {
      if ( (root.lRef.snum == sym.ContractNode) ||
           (root.lRef.snum == sym.ContractEdge) )
        return;
      if ( (root.lRef.snum == sym.ExtendNode) || (root.lRef.snum ==
sym.ExtendEdge) )
      {
        update_map(tempmap, scheme_to_name(root.rRef)+num,
emptyQuery);
        return;
      }
      String s = scheme_to_name(root.lRef.rRef);
      int isnum = root.lRef.lRef.snum;
      if ( (isnum == sym.AddNode) || (isnum == sym.AddEdge) )
        update_map(tempmap, s+num, root.rRef);
      else if (isnum == sym.RenameNode)
        text_replace(tempmap, cleanStr(root.rRef.lRef.textVal), s);
      else if (isnum == sym.RenameEdge)
        text_replace(tempmap,
cleanStr(root.rRef.lRef.textVal)+s.substring(s.indexOf('_')), s );
      return;
    }
    process_tran(root.rRef, tempmap, num); //Note unusual way round.
    process_tran(root.lRef, tempmap, num);
  }

  private void buildSubMap(Node root)
  {
    if (root==null) return;
    if (root.snum == Elements.transformation)
    {
      String num = root.lRef.lRef.rRef.textVal;
        // num will hold the number of the local schema being worked
with.

      HashMap tempmap = new HashMap();
      for (int i=0; i<globalnames.size(); i++)
      {
        tempmap.put(globalnames.get(i), new Node(Elements.query, new
Node(
          Elements.expr, new Node(sym.VarToken,
(String)(globalnames.get(i))+num ), null) , null) );

      }
      process_tran(root, tempmap, num);
      for (int i=0; i<globalnames.size(); i++)
      {
        Node q1 = (Node)(mainMap.get(globalnames.get(i)));
        Node q2 = (Node)(tempmap.get(globalnames.get(i)));
        Node nq = new Node(Elements.query,
          new Node(q1, new Node(sym.Append, "++") ), q2);
        mainMap.put(globalnames.get(i), nq.copySubtree());
      }

      return;
    }
    buildSubMap(root.lRef);
    buildSubMap(root.rRef);
  }

  public void performSubs(Node troot)
  {
    if (troot == null) return;
    if (troot.snum == Elements.expr)
      if (troot.lRef.snum == sym.VarToken)
      {
        Node n = (Node)(mainMap.get(troot.lRef.textVal));
        if (n != null)
          troot.copyNode( new Node(Elements.expr, new Node(
            new Node(sym.LRB, "("), n), new Node(sym.RRB, ")") ) );
        return;
      }
    performSubs(troot.lRef);
    performSubs(troot.rRef);
  }

  public Substituter(Node root)
  {
    mainMap = new HashMap();
    globalnames = new ArrayList();

    buildGlobalNames(root.lRef);

    for (int i=0; i<globalnames.size(); i++)
      mainMap.put(globalnames.get(i), emptyQuery.copySubtree());

    buildSubMap(root.rRef);
  }
}
```

Simplifier.java

```
package QTran;

import java.util.*;

public class Simplifier
{
  ArrayList querypairs;

  static final String startVarStr = "xxx";

  private class QueryPair
  {
    Node from, to;

    QueryPair(Node f, Node t)
    {
      from = f.copySubtree();
      to = t.copySubtree();
    }
  }

  private boolean comp(Node queryNd, Node templateNd, HashMap mapv)
  {
    if ((queryNd==null)||(templateNd==null))
    {
      if ((queryNd==null)&&(templateNd==null)) return true;
      else return false;
    }
    if ((templateNd.textVal!=null) &&
(templateNd.textVal.startsWith(startVarStr)))
    {
      mapv.put(templateNd.textVal, queryNd);
      return true;
    }
    if (queryNd.snum!=templateNd.snum)
      return false;
    if (queryNd.textVal!=null)
    {
      if (!(queryNd.textVal.equals(templateNd.textVal))) return false;
    }
    else if (templateNd.textVal!=null) return false;
    boolean b1 = comp(queryNd.lRef, templateNd.lRef, mapv);
    boolean b2 = comp(queryNd.rRef, templateNd.rRef, mapv);
    return (b1&&b2);
  }

  private void replace(Node root, HashMap mapv)
  {
    if (root==null) return;
    if ((root.textVal!=null) &&
(root.textVal.startsWith(startVarStr)))
    {
      Node n = (Node)(mapv.get(root.textVal));
      root.copyNode(n);
      return;
    }
    replace(root.lRef, mapv);
    replace(root.rRef, mapv);
  }
```

```
private boolean compare(Node root, int i)
{
  HashMap mapvar = new HashMap();
  QueryPair qp = (QueryPair)(querypairs.get(i));
  boolean b = comp(root, qp.from, mapvar);
  if (b)
  {
    root.copyNode( qp.to );
    replace(root, mapvar);
  }
  return b;
}

private boolean check(Node root)
{
  boolean b = false;
  for(int i=0; i<querypairs.size(); i++)
  {
    boolean b2 = compare(root, i);
    b = b||b2;
  }
  return b;
}

private boolean simp(Node root)
{
  if (root==null) return false;
  boolean a,b,c;
  a = check(root);
  b = simp(root.lRef);
  c = simp(root.rRef);
  return a||b||c;
}

public void simplify(Node root)
{
  boolean b;
  do
    b = simp(root);
  while (b);
}

private void cleanTemplate(Node root)
{
  if (root==null) return;
  if ( (root.snum==Elements.query) &&
       (root.lRef.snum==Elements.expr) &&
       (root.lRef.lRef.snum==sym.VarToken) &&
       (root.lRef.lRef.textVal.startsWith(startVarStr) ) )
  {
    root.textVal = root.lRef.lRef.textVal;
    return;
  }
  cleanTemplate(root.lRef);
  cleanTemplate(root.rRef);
}

private void cleanTemplates()
{
  for(int i=0; i<querypairs.size(); i++)
```

```
    {
      cleanTemplate( ((QueryPair)(querypairs.get(i))).from );
      cleanTemplate( ((QueryPair)(querypairs.get(i))).to );
    }
  }

  private void setItUp(Node root)
  {
    if (root==null) return;
    if (root.snum == Elements.simplification)
    {
      querypairs.add( new QueryPair( root.lRef.lRef.rRef, root.rRef )
);
      return;
    }
    setItUp(root.lRef);
    setItUp(root.rRef);
  }

  public Simplifier(Node root)
  {
    querypairs = new ArrayList();
    setItUp(root);
    cleanTemplates();
  }
}
```

Elements.java

```
package QTran;

public class Elements
{
  public static final int PARTIAL_STRUCTURE = 999;
  public static final int seq = 1004;
  public static final int qual = 1006;
  public static final int quals = 1005;
  public static final int query = 1002;
  public static final int expr = 1003;
  public static final int name = 1007;
  public static final int transformation = 1008;
  public static final int transf_list = 1009;
  public static final int prim_transf = 1010;
  public static final int scheme = 1011;
  public static final int nodeScheme = 1012;
  public static final int edgeScheme = 1013;
  public static final int scheme_list = 1014;
  public static final int schema_def = 1015;
  public static final int schema_defs = 1016;
  public static final int transformations = 1017;
  public static final int allinfo = 1018;
  public static final int simplification = 1019;
  public static final int simp_list = 1020;
}
```

TestDriver.java

```
package QTran;

import java.io.*;

public class TestDriver
{
  private static final char LOADTRANS = 'a';
  private static final char LISTSUBS = 'b';
  private static final char GETQUERY = 'c';
  private static final char PRINTQUERY = 'd';
  private static final char PRINTTREE = 'e';
  private static final char TRANSLATE = 'f';
  private static final char LOADSIMPS = 'g';
  private static final char SIMPLIFY = 'h';
  private static final char EXIT = 'x';

  Substituter sub;
  Simplifier simp;
  Node q;

  TestDriver() {}

  public static void main(String[] args) throws Exception
  {
    TestDriver d = new TestDriver();
    do {
      System.out.println("\t\tMenu");
      System.out.println("\t" + LOADTRANS + "- load transformations
file");
      System.out.println("\t" + LISTSUBS + "- display substitutions
made during translation");
      System.out.println("\t" + GETQUERY + "- load query from file");
      System.out.println("\t" + PRINTQUERY + "- print query");
      System.out.println("\t" + PRINTTREE + "- output text
representation of query tree");
      System.out.println("\t" + TRANSLATE + "- translate query");
      System.out.println("\t" + LOADSIMPS + "- load simplification
file");
      System.out.println("\t" + SIMPLIFY + "- perform
simplifications");
      System.out.println("\t" + EXIT + "- exit");
      System.out.print("\t>");
    } while ( d.action(TextIO.getlnChar()));
    System.out.println("\n\nFinished!");
  }


  private boolean action(char c) throws Exception
  {
    switch(c){
      case LOADTRANS : loadTrans(); break;
      case LISTSUBS : listSubs(); break;
      case GETQUERY : getQuery(); break;
      case PRINTQUERY : printIt(); break;
      case PRINTTREE : printTree(); break;
      case TRANSLATE : translate(); break;
      case LOADSIMPS : loadSimps(); break;
      case SIMPLIFY : simplify(); break;
      case EXIT : return false;
      default: System.out.println("** Unknown selection [" + c + "]
**");
    }
    return true;
  }

  private void simplify()
  {
    simp.simplify(q);
  }

  private void translate()
  {
    sub.performSubs(q);
  }

  private void printIt()
  {
    printQuery(q);
    System.out.println();
  }

  private void printTree()
  {
    outputTree(q);
    System.out.println();
  }

  private void getQuery() throws Exception
  {
    System.out.print("Enter the file name: ");
    String name = TextIO.getlnString();
    q = new Node(new FileInputStream(name));
  }

  private void loadSimps() throws Exception
  {
    System.out.print("Enter the file name: ");
    String name = TextIO.getlnString();
    Node n = new Node(new FileInputStream(name));
    simp = new Simplifier(n);
  }

  private void loadTrans() throws Exception
  {
    System.out.print("Enter the file name: ");
    String name = TextIO.getlnString();
    Node n = new Node(new FileInputStream(name));
    sub = new Substituter(n);
  }

  private void listSubs()
  {
    for (int i=0; i<sub.globalnames.size(); i++)
    {
      String s = (String)(sub.globalnames.get(i));
      System.out.println(s);
      printQuery((Node)(sub.mainMap.get(s)));
      System.out.println();
    }
  }
  public static void outputTree(Node n)
  {
    System.out.print(n.snum + "|");
    if (n.textVal!=null) System.out.print(n.textVal);
    System.out.print("{");
    if (n.lRef!=null) outputTree(n.lRef);
    System.out.print(",");
    if (n.rRef!=null) outputTree(n.rRef);
    System.out.print("}");
  }

  public static void printQuery(Node n)
  {
    if (n==null) return;
    printQuery(n.lRef);
    if (n.textVal!=null) System.out.print(n.textVal+" ");
    printQuery(n.rRef);
  }
}
```

APPENDIX B – CUP and JLex Files

EdCup1.cup

```
package QTran;

terminal VarToken, StrToken, NumToken, AnyToken,
  Let, Equal, Append, Difference, In, SemiColon, LArrow,
  Comma, Bar, LSB, RSB, LRB, RRB,
  LDAB, RDAB, FromSchema, End,
  AddNode, DelNode, AddEdge, DelEdge, RenameNode, RenameEdge,
  ExtendNode, ContractNode, ExtendEdge, ContractEdge,
  From, To;

non terminal query, expr, seq, quals, qual,
  name, transformation, transf_list, prim_transf, scheme,
  nodeScheme, edgeScheme, scheme_list, transformations,
  simplification, simp_list,
  allinfo;

precedence left Let, Equal, Append, Difference, In, LArrow, SemiColon,
  Comma, Bar, LSB, RSB, LRB, RRB, LDAB, RDAB, FromSchema, End,
  AnyToken, VarToken, NumToken, StrToken;

allinfo ::=   scheme_list:e1 transformations:e2
              {: RESULT = new Node(Elements.allinfo, e1, e2);
              :}
            | query:e1
              {: RESULT = new Node(Elements.allinfo, e1, null);
              :}
            | simp_list:e1
              {: RESULT = new Node(Elements.allinfo, e1, null);
              :}
              ;

simp_list ::=   simp_list:e1 simplification:e2
                {: RESULT = new Node(Elements.simp_list, e1, e2);
                :}
              | simplification:e1
                {: RESULT = new Node(Elements.simp_list, e1, null);
                :}
                ;

simplification ::= From:e1 query:e2 To:e3 query:e4
                   {: RESULT = new Node(Elements.simplification,
                       new Node(new Node(e1, e2), e3), e4);
                   :}
                   ;

transformations ::= transformations:e1 transformation:e2
                    {: RESULT = new Node(Elements.transformations,
                      e1, e2);
                    :}
                  | transformation:e1
                    {: RESULT = new Node(Elements.transformations,
                      e1, null);
                    :}
                    ;

transformation ::= FromSchema:e1 NumToken:e2 transf_list:e3 End:e4
```

```
                    {:
                      RESULT = new Node(Elements.transformation,
                        new Node(new Node(e1, e2), e3), e4);
                    :}
                    ;

transf_list    ::=    prim_transf:e1 SemiColon:e2 transf_list:e3
                      {: RESULT = new Node(Elements.transf_list,
                        new Node(e1, e2), e3);
                      :}
                    | prim_transf:e1
                      {: RESULT = new Node(Elements.transf_list,
                        e1, null);
                      :}
                      ;

prim_transf    ::=    AddNode:e1 nodeScheme:e2 query:e3
                      {: RESULT = new Node(Elements.prim_transf,
                        new Node(e1, e2), e3);
                      :}
                    | DelNode:e1 nodeScheme:e2 query:e3
                      {: RESULT = new Node(Elements.prim_transf,
                        new Node(e1, e2), e3);
                      :}
                    | AddEdge:e1 edgeScheme:e2 query:e3
                      {: RESULT = new Node(Elements.prim_transf,
                        new Node(e1, e2), e3);
                      :}
                    | DelEdge:e1 edgeScheme:e2 query:e3
                      {: RESULT = new Node(Elements.prim_transf,
                        new Node(e1, e2), e3);
                      :}

                    | RenameNode:e1 nodeScheme:e2 name:e3
                      {: RESULT = new Node(Elements.prim_transf,
                        new Node(e1, e2), e3);
                      :}

                    | RenameEdge:e1 edgeScheme:e2 name:e3
                      {: RESULT = new Node(Elements.prim_transf,
                        new Node(e1, e2), e3);
                      :}

                    | ExtendNode:e1 nodeScheme:e2
                      {: RESULT = new Node(Elements.prim_transf, e1,
e2);
                      :}

                    | ContractNode:e1 nodeScheme:e2
                      {: RESULT = new Node(Elements.prim_transf, e1,
e2);
                      :}

                    | ExtendEdge:e1 edgeScheme:e2
                      {: RESULT = new Node(Elements.prim_transf, e1,
e2);
                      :}

                    | ContractEdge:e1 edgeScheme:e2
                      {: RESULT = new Node(Elements.prim_transf, e1,
e2);
```

```
                      :}
                    ;

scheme  ::=   nodeScheme:e1
              {:
                RESULT = new Node(Elements.scheme, e1, null);
              :}
            | edgeScheme:e1
              {:
                RESULT = new Node(Elements.scheme, e1, null);
              :}
              ;

nodeScheme ::= LDAB:e1 name:e2 RDAB:e3
               {:
                 RESULT = new Node(Elements.nodeScheme,
                   new Node(e1, e2), e3);
               :}
               ;

edgeScheme ::= LDAB:e1 name:e2 Comma:e3 scheme_list:e4 RDAB:e5
               {:
                 RESULT = new Node(Elements.edgeScheme,
                   new Node(new Node(new Node(e1, e2), e3), e4), e5);
               :}
               ;

scheme_list ::=   scheme:e1
                  {: RESULT =
                    new Node(Elements.scheme_list, e1, null);
                  :}
                | scheme_list:e1 Comma:e2 scheme:e3
                  {: RESULT = new Node(Elements.scheme_list,
                    new Node(e1, e2), e3);
                  :}
                  ;

name ::= StrToken:e1
         {: RESULT = new Node(Elements.name, e1, null); :};

query ::=   expr:e1
            {: RESULT =
                 new Node(Elements.query, e1, null);
            :}
          | Let:e1 VarToken:e2 Equal:e3 query:e4 In:e5 query:e6
            {: RESULT = new Node(Elements.query,
                 new Node( new Node( new Node(
                   new Node(e1, e2), e3), e4), e5), e6);
            :}
          | query:e1 Append:e2 query:e3
            {: RESULT = new Node(Elements.query,
                 new Node(e1, e2), e3);
            :}
          | query:e1 Difference:e2 query:e3
            {: RESULT = new Node(Elements.query,
                 new Node(e1, e2), e3);
            :}
          | query:e1 expr:e2
            {: RESULT = new Node(Elements.query, e1, e2);
            :};
```

```
expr ::=    NumToken:e1
            {: RESULT = new Node(Elements.expr, e1, null); :}
          | StrToken:e1
            {: RESULT = new Node(Elements.expr, e1, null); :}
          | VarToken:e1
            {: RESULT = new Node(Elements.expr, e1, null); :}
          | AnyToken:e1
            {: RESULT = new Node(Elements.expr, e1, null); :}
          | LSB:e1 query:e2 Bar:e3 quals:e4 RSB:e5
            {: RESULT = new Node(Elements.expr,
                 new Node( new Node( new Node(e1, e2), e3), e4), e5);
            :}
          | LSB:e1 RSB:e2
            {: RESULT = new Node(Elements.expr, e1, e2); :}
          | LRB:e1 seq:e2 RRB:e3
            {: RESULT = new Node(Elements.expr,
                 new Node(e1, e2), e3);
            :}
          | LRB:e1 query:e2 RRB:e3
            {: RESULT = new Node(Elements.expr, new Node(e1, e2), e3);
            :};

seq ::=     seq:e1 Comma:e2 query:e3
            {: RESULT = new Node(Elements.seq, new Node(e1, e2), e3);
            :}
          | query:e1
            {: RESULT = new Node(Elements.seq, e1, null); :}
            ;

quals ::=   qual:e1 SemiColon:e2 quals:e3
            {: RESULT = new Node(Elements.quals, new Node(e1, e2),
e3);
            :}
          | qual:e1
            {: RESULT = new Node(Elements.quals, e1, null); :}
            ;

qual ::=    query:e1
            {: RESULT = new Node(Elements.qual, e1, null); :}
          | query:e1 LArrow:e2 query:e3
            {: RESULT = new Node(Elements.qual, new Node(e1, e2), e3);
            :}
            ;
```

EdLex1

```
package QTran;

import java_cup.*;
import java_cup.runtime.*;

public class EdLex1
{
  public static void main(String[] args) throws Exception
  {
    Yylex yy = new Yylex(System.in);
    Symbol t;
    while (( t = yy.next_token() ).sym != sym.EOF )
      System.out.println(t + " " + ((Node)(t.value)).textVal);
  }
}

%%

%cup

%eofval{
return (new Symbol(sym.EOF,""));
%eofval}

NUMTOKEN = [0-9]+|([0-9]+)(".")([0-9]+)
STRTOKEN = \"[^\"]*\"
PREFIXOPERATOR = "(+)"|"(-
)"|"(*)"|"(/)"|"(&)"|"(#)"|"(=)"|"(!=)"|"(<)"|"(>)"|"(<=)"|"(>=)"
VARTOKEN = [a-z_][A-Za-z0-9_]*

NN_WHITESPACE = [\ \t\b\012]+

%%

"let" { return (new Symbol(sym.Let, new Node(sym.Let, "let"))); }
"in"  { return (new Symbol(sym.In, new Node(sym.In, "in"))); }
"="   { return (new Symbol(sym.Equal, new Node(sym.Equal, "="))); }
"++"  { return (new Symbol(sym.Append, new Node(sym.Append, "++"))); }
"--"  { return (new Symbol(sym.Difference, new Node(sym.Difference, "-
-"))); }
";"   { return (new Symbol(sym.SemiColon, new Node(sym.SemiColon,
";"))); }
"<-"  { return (new Symbol(sym.LArrow, new Node(sym.LArrow, "<-"))); }
","   { return (new Symbol(sym.Comma, new Node(sym.Comma, ","))); }
"|"   { return (new Symbol(sym.Bar, new Node(sym.Bar, "|"))); }
"["   { return (new Symbol(sym.LSB, new Node(sym.LSB, "["))); }
"]"   { return (new Symbol(sym.RSB, new Node(sym.RSB, "]"))); }
"("   { return (new Symbol(sym.LRB, new Node(sym.LRB, "("))); }
")"   { return (new Symbol(sym.RRB, new Node(sym.RRB, ")"))); }
"Any" { return (new Symbol(sym.AnyToken, new Node(sym.AnyToken,
"Any"))); }

"<<"  { return (new Symbol(sym.LDAB, new Node(sym.LDAB, "<<"))); }
">>"  { return (new Symbol(sym.RDAB, new Node(sym.RDAB, ">>"))); }

"FromSchema"  { return (new Symbol(sym.FromSchema, new
Node(sym.FromSchema, "FromSchema"))); }
"End"         { return (new Symbol(sym.End, new Node(sym.End,
"End"))); }
```

```
"addNode" { return (new Symbol(sym.AddNode, new Node(sym.AddNode,
yytext() ) ) ); }
"delNode" { return (new Symbol(sym.DelNode, new Node(sym.DelNode,
yytext() ) ) ); }
"addEdge" { return (new Symbol(sym.AddEdge, new Node(sym.AddEdge,
yytext() ) ) ); }
"delEdge" { return (new Symbol(sym.DelEdge, new Node(sym.DelEdge,
yytext() ) ) ); }
"renameNode"   { return (new Symbol(sym.RenameNode, new
Node(sym.RenameNode, yytext() ) ) ); }
"renameEdge"   { return (new Symbol(sym.RenameEdge, new
Node(sym.RenameEdge, yytext() ) ) ); }
"extendNode"   { return (new Symbol(sym.ExtendNode, new
Node(sym.ExtendNode, yytext() ) ) ); }
"contractNode" { return (new Symbol(sym.ContractNode, new
Node(sym.ContractNode, yytext() ) ) ); }
"extendEdge"   { return (new Symbol(sym.ExtendEdge, new
Node(sym.ExtendEdge, yytext() ) ) ); }
"contractEdge" { return (new Symbol(sym.ContractEdge, new
Node(sym.ContractEdge, yytext() ) ) ); }

"From" { return (new Symbol(sym.From, new Node(sym.From, "From"))); }
"To"   { return (new Symbol(sym.To, new Node(sym.To, "To"))); }

{PREFIXOPERATOR} { return (new Symbol(sym.VarToken, new
Node(sym.VarToken, yytext() ) ) ); }
{VARTOKEN} { return (new Symbol(sym.VarToken ,new Node(sym.VarToken,
yytext() ) ) ); }
{STRTOKEN} { return (new Symbol(sym.StrToken ,new Node(sym.StrToken,
yytext() ) ) ); }
{NUMTOKEN} { return (new Symbol(sym.NumToken ,new Node(sym.NumToken,
yytext() ) ) ); }

{NN_WHITESPACE} { }

. { return (new Symbol(sym.error,"**ERROR**") ); }

\n { }
\r { }
```

APPENDIX C – Text Files

QFile1.txt

```
group(_person_qualification)
```

TFile1.txt

```
<<"person">>,
<<"address">>,
<<"qualification">>,
<<"post">>,
<<"",<<"person">>,<<"address">> >>,
<<"",<<"person">>,<<"qualification">> >>,
<<"",<<"person">>,<<"post">> >>
FromSchema 1
addNode <<"person">> men1 ++ women1;
addEdge <<"", <<"person">>, <<"post">> >> _men_post1 ++_women_post1;
contractEdge <<"", <<"men">>, <<"post">> >>;
contractEdge <<"", <<"women">>, <<"post">> >>;
contractNode <<"men">>;
contractNode <<"women">>;
extendNode <<"address">>;
extendEdge <<"",<<"person">>,<<"address">> >>;
extendNode <<"qualification">>;
extendEdge <<"",<<"person">>,<<"qualification">> >>
End
FromSchema 2
renameNode <<"name">> "person";
extendNode <<"post">>;
extendEdge <<"",<<"person">>,<<"post">> >>
End
FromSchema 3
renameEdge <<"quals", <<"person">>, <<"qualification">> >> "";
extendNode <<"address">>;
extendEdge <<"", <<"person">>, <<"address">> >>
End
```

SFile1.txt

```
From group(xxx1 ++ xxx2) To (group(xxx1)) merge (group(xxx2))
From xxx1 ++ [] To xxx1
From [] ++ xxx1 To xxx1
From ([]) To []
From ((xxx1)) To (xxx1)
From xxx1 ((xxx2)) To xxx1 (xxx2)
```

SFile2.txt

```
From xxx1 ++ [] To xxx1
From [] ++ xxx1 To xxx1
From ([]) To []
From ((xxx1)) To (xxx1)
From xxx1 ((xxx2)) To xxx1 (xxx2)
```

SFile3.txt

```
From group(xxx1 ++ xxx2) To (group(xxx1)) merge (group(xxx2))
```

APPENDIX D – Running Instructions

1) Copy the directories QTran, java_cup and JLex and their contents somewhere into the Java class path on the machine.

2) The test driver can then be executed by entering:
```
java QTran.TestDriver
```

3) Queries, simplifications and transformations can be loaded from text files using the menu. The test driver only deals with one query, one set of transformations and one set of simplifications at a time. Loaded a new simplification file, for instance, removes the old one – but it doesn't remove the current query or set of transformations.