EXPRESS QUERY LANGUAGE AND TEMPLATES AND RULES:
TWO LANGUAGES FOR ADVANCED SOFTWARE SYSTEM INTEGRATIONS

A Dissertation Presented to

The Faculty of the

Fritz J. and Dolores H. Russ
College of Engineering and Technology
Ohio University

In Partial Fulfillment

of the Requirement for the Degree

Doctor of Philosophy

By

Lizhong Huang

June, 1999
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.1</td>
<td>ISO 10303-11</td>
<td>29</td>
</tr>
<tr>
<td>2.2.2</td>
<td>ISO 10303-21</td>
<td>31</td>
</tr>
<tr>
<td>2.2.3</td>
<td>ISO 10303-22</td>
<td>32</td>
</tr>
<tr>
<td>2.3</td>
<td>Examples Used In This Chapter</td>
<td>32</td>
</tr>
<tr>
<td>2.4</td>
<td>Overview and Glossaries</td>
<td>38</td>
</tr>
<tr>
<td>2.5</td>
<td>From Clause</td>
<td>41</td>
</tr>
<tr>
<td>2.6</td>
<td>Where Clause</td>
<td>43</td>
</tr>
<tr>
<td>2.6.1</td>
<td>NULL Condition</td>
<td>44</td>
</tr>
<tr>
<td>2.6.2</td>
<td>EMPTY Condition</td>
<td>45</td>
</tr>
<tr>
<td>2.6.3</td>
<td>ISA Condition</td>
<td>46</td>
</tr>
<tr>
<td>2.6.4</td>
<td>Comparison Condition</td>
<td>46</td>
</tr>
<tr>
<td>2.6.5</td>
<td>IN Condition</td>
<td>48</td>
</tr>
<tr>
<td>2.6.6</td>
<td>CONTAINS Condition</td>
<td>49</td>
</tr>
<tr>
<td>2.6.7</td>
<td>MATCH Condition</td>
<td>49</td>
</tr>
<tr>
<td>2.6.8</td>
<td>FOR Condition</td>
<td>50</td>
</tr>
<tr>
<td>2.6.9</td>
<td>Compound Condition</td>
<td>51</td>
</tr>
<tr>
<td>2.7</td>
<td>Select Query</td>
<td>51</td>
</tr>
<tr>
<td>2.8</td>
<td>Update Query</td>
<td>53</td>
</tr>
<tr>
<td>2.9</td>
<td>Insert Query</td>
<td>55</td>
</tr>
<tr>
<td>2.10</td>
<td>Delete Query</td>
<td>57</td>
</tr>
<tr>
<td>2.11</td>
<td>System Commands</td>
<td>58</td>
</tr>
</tbody>
</table>
3 An Implementation of EXPRESS Query Language

3.1 Introduction ................................................................. 60
3.2 Overview ...................................................................... 61
3.3 Data Structures ............................................................ 63
3.4 Major Node Constructs .................................................. 68
  3.4.1 Variable Declaration .................................................. 68
  3.4.2 Element Node ............................................................ 70
  3.4.3 Attribute Node .......................................................... 70
  3.4.4 Inverse Attribute Node .............................................. 71
  3.4.5 Condition Nodes ....................................................... 71
  3.4.6 For-Condition Node ................................................ 73
  3.4.7 Select Expression Nodes .......................................... 74
3.5 Significant Global Variables .......................................... 74
3.6 Major Routines ............................................................. 77
  3.6.1 Manipulation of a Q_NODE ........................................ 77
  3.6.2 Manipulation of a V_NODE ........................................ 78
  3.6.3 Manipulation of Host Variable Nodes ......................... 78
  3.6.4 Binding Q NODES .................................................. 78
  3.6.5 Execution of Q NODES .............................................. 78
3.7 Model Interfaces ........................................................... 79
3.8 Caveats ......................................................................... 81
3.9 How To Use Eql Interactive Program ........................................... 83

3.9.1 Setting Up Environment Variables ........................................ 83
3.9.2 Compilation of EXPRESS schema ......................................... 83
3.9.3 Launching Eql Interactive Program ..................................... 84

4 The Templates and Rules Language ........................................... 85

4.1 Introduction ........................................................................... 85
4.2 Overview .............................................................................. 86
4.3 Variables .............................................................................. 86

4.3.1 Declaration and Scope .................................................... 87
4.3.2 Memory Storage Management .......................................... 88

4.4 Operators and Expressions .................................................. 88

4.4.1 Arithmetic Operators ....................................................... 89
4.4.2 Comparison Operators ...................................................... 90
4.4.3 Logical Operators ........................................................... 90
4.4.4 String Operator ............................................................... 91
4.4.5 Assigning Operators ......................................................... 91
4.4.6 Other Operators .............................................................. 91

4.5 Control Flow ........................................................................ 92

4.5.1 If Statement .................................................................... 92
4.5.2 Loops .............................................................................. 94
4.5.3 Switch Statement ............................................................. 95
4.5.4 Break and Continue ...................................................... 96

4.6 Procedures ........................................................................ 96

4.6.1 Calling a Procedure ..................................................... 96

4.6.2 Bring Command Line Arguments into Program .............. 98

4.7 Data Analysis ..................................................................... 99

4.7.1 Mathematical functions .................................................. 99

4.7.2 String Manipulations ..................................................... 100

4.7.3 Others ........................................................................ 103

4.8 Accessing Data Resources .................................................. 104

4.8.1 Files ........................................................................ 105

4.8.2 Regular Related Functions ............................................. 106

4.8.3 Databases ................................................................... 108

4.8.4 Host Variables ............................................................. 109

4.8.5 STEP files ................................................................... 110

4.8.6 Networks ..................................................................... 111

4.8.7 Transient Data Access .................................................. 111

4.9 Miscellaneous .................................................................... 111

4.9.1 Literal ........................................................................... 111

4.9.2 Comments ..................................................................... 112

4.9.3 Environment Variable ................................................... 112

4.9.4 Process .......................................................................... 112

4.9.5 Modularization of Program ............................................ 113
5 The Implementation of TNR Language

5.1 Overview ................................................................. 115

5.2 Internal Data Structures ............................................... 116
  5.2.1 Symbol Node ..................................................... 116
  5.2.2 Parse Node ...................................................... 119
  5.2.3 Variable ........................................................... 120

5.3 Major Parse Node Constructs ......................................... 121
  5.3.1 Statement Nodes ................................................ 122
  5.3.2 Types of Statement Nodes ...................................... 122
  5.3.3 Types of Expression Nodes .................................... 124
  5.3.4 Typical Function Nodes ....................................... 127
  5.3.5 Select Rule Nodes .............................................. 128

5.4 Significant Global Variables ........................................... 128

5.5 Implementation Algorithms and Major Routines ........................ 130
  5.5.1 Manipulation of Symbol Nodes ................................ 130
  5.5.2 Manipulation of Variables .................................... 131
  5.5.3 Manipulation of Parse Nodes .................................. 132
  5.5.4 Manipulations of Multiple Templates and Rules
      Programs ............................................................. 132
  5.5.5 File Stacks ...................................................... 133
  5.5.6 I/O functions ................................................... 134
  5.5.7 Procedure Calls ............................................... 135
5.5.8 Interface of SQL .......................................................... 136
5.5.9 Interface to Eql .......................................................... 138

6 Software Integration Using Templates and Rules and EXPRESS Query Language

6.1 Introduction ................................................................. 139
6.2 Example One ............................................................... 139
   6.2.1 Data Translation From STEP To Oracle ...................... 142
   6.2.2 Report Generation ................................................. 146
6.3 Example Two ............................................................... 152
   6.3.1 IMW & IMDE ....................................................... 152
   6.3.2 UDMM .............................................................. 153
   6.3.3 Implementation of IMW Using Templates and Rules
       and EXPRESS Query Language ......................... 154

7 Conclusions ................................................................. 156
   7.1 EXPRESS Query Language ....................................... 157
   7.2 Templates and Rules .............................................. 158

Bibliography ................................................................. 160

Abstract
List of Tables

2.1 Comparable Data Types(*: for == and != only; **: for All

  Comparison Operators; Otherwise, No Comparison Defined)..................47

3.1 Unknown Values For Some Data Types........................................82

4.1 Special Characters.....................................................................89

4.2 Operators Precedence.................................................................92

4.3 Regular Expressions For Match Condition (this table is adapted

  from UNIX manual page in IRIX 5.3)............................................101
List of Figures

1.1 Architecture for the Intelligent Machining Workstation .......................... 11
1.2 Intelligent Interface of IMDE .................................................................... 12
1.3 A typical data translator implemented with Tnr ........................................ 23
2.1 Grammar rules for constant values ........................................................ 41
2.2 Grammar rules for the FROM-statement .............................................. 43
2.3 The syntax definition FOR conditions .................................................. 44
2.4 The syntax definition for NULL conditions ............................................ 45
2.5 The syntax definition for EMPTY conditions ......................................... 46
2.6 The syntax definition for ISA conditions ............................................... 46
2.7 The syntax definition for comparison conditions .................................... 47
2.8 The syntax definition for IN conditions ................................................ 49
2.9 The syntax definition for CONTAINS conditions .................................... 49
2.10 The syntax definition for MATCH conditions ....................................... 50
2.11 The syntax definition for FOR conditions .......................................... 50
2.12 The syntax definition for SELECT queries .......................................... 51
2.13 The syntax definition for an UPDATE query ....................................... 54
2.14 The syntax definition for INSERT query ............................................. 56
2.15 The syntax definition for DELETE queries .......................................... 57
3.1 EQL has three interpretation stages ...................................................... 61
3.2 The architecture of EQL ........................................................................ 62
3.3 The backbone of the EQL node tree (portion shadowed is not supported by the current version of EQL).................................................................65

3.4 An example Eql query.................................................................................66

3.5 Node tree constructed from the example query in Figure 3.4......................67

3.6 Nodes for variable declarations.................................................................69

3.7 Nodes for conditions................................................................................72

3.8 Nodes for for conditions...........................................................................73

3.9 Select expressions....................................................................................75

3.10 EQL prompts "eql>" and waits for user input...........................................84

5.1 Tnr has three interpretation stages............................................................115

5.2 The node tree of a Tnr program ...............................................................118

5.3 Variable data structure................................................................................121

5.4 Some statement node constructs...............................................................125

5.5 Node constructs for a procedure call statement.........................................127

5.6 Node construct for a built-in function.......................................................128

5.7 Node construct for select_rule() function...............................................129

5.8 Procedure call...........................................................................................137

6.1 Data migration using Tnr programs...........................................................140
1.1 Introduction

Integration is beautiful yet difficult to achieve. Its beauty is reflected in the "unified whole" which works cooperatively and harmoniously; its difficulty comes from the coordination of the numerous internal units. As with many technological processes today, it can be aided by computers. Computer-based integration employs computers and computer software to achieve the integration process. There are many computer-integrated systems today, such as computer integrated information systems, integrated manufacturing systems, etc. [11][36][38]

Computer integrated systems call for much more than the installation of new and expensive hardware; they also require the integration of software[34][1][6]. Software integration can be defined as two or more stand alone software systems which work together as a seamless package and yield more powerful functionalities than each would individually. Usually, software integration has the following characteristics:
1. Each relevant individual software is stand-alone and has its well defined working domain. The individual software itself may contain numerous software components, but the connections among them are seamless.

2. The sub-software systems are related in the sense that their functionality domains intersect. Therefore, they usually have data dependency and/or data sharing relationships. For example, the output of a part geometry from a CAD tool can be the partial input of a CAM tool.

3. The sub-systems are usually heterogeneous. That means they may have been implemented and now operate in different environments. The environments include different hardware and different operating systems. They may also have different data storage methods.

4. The sub-systems may reside in different geographical locations. Modern enterprises usually have many branches in different places. Integration of these systems, therefore, need to rely on a computer network.

   One of the most important applications of software integration is the construction of computer-integrated systems. Today's engineering systems are getting more and more complex, and a single software package can hardly meet all the designing requirements for a whole system. In practice, software vendors usually focus their products only on a few sub-domains. These small-to-medium-scale systems are much more controllable on the both development and maintaining stages. However, the compatibility and interoperability of these systems are challenging because of the difference of data modeling and the lack of API standards. For example, it has been observed that manufacturing engineering is an area that has not been an area served well
by computer systems over the last decade[34]. A manufacturing system consists of
design, process planning, cost evaluation, inventory controlling, etc. and several
procedural applications. The applications (software) inside a system do not inter-
operate. Therefore, extensive human participation is needed, which is not only error-
prone but also time-consuming.

Secondly, software integration can greatly increase the reusability of software.
Since the beginning of the software history, software reusability has been an important
issue. Subroutines, function libraries and standard class libraries are all means for
increasing software reusability. All these methods are based on reusable modules which
can be modified as source codes. Software components[2][27] is an emerging
technology that increases reusability by providing reusable modules that can be
modified at design time as executable modules. Software integration achieves this goal
by going a step further. It interconnects numerous stand-alone software systems as
monolithic modules without any modification and produces new systems beyond the
summation of the functions of the individual software systems.

Finally, software integration helps to bridge the “islands of information”. Computer networks physically make a huge amount of information easier to access
instead of bringing up new computations or data. However, the heterogeneous
characteristics of the software systems over the network limit their communication to a
high degree[41]. With software integration technology, the systems can share their
information more easily and more efficiently.
1.2 Levels of Software Integration

In practice, software integration is not always in full scale. The ingredients of software integration can be classified into three levels independent of the implementation methods. These three levels are: data sharing, data associativity, and events sharing.

- **Data sharing** means that several systems share their proprietary data with one another. The sharing activities consist of both reading and writing. Data sharing is passive; in another words, if the data are changed by one party other parties will not notice the change until the next time they access the data. Data sharing provides the fundamental provision for software integration.

- **Data associativity** makes tighter data linkage among software systems in a sense that sharing data becomes active. That means when data are changed by one party, all other parties concerned will react to the change instantly. The reaction includes all the necessary recalculation related to those data. For example, the new feature called "parametric design" of the new generation CAD tools has this level integration. This feature makes the change of the dimension of a sketch in a sketch note propagate to solid models automatically. Data associativity brings higher level of data integration than does data sharing.

- **Event sharing** makes it possible for different systems to share the same events. An event is a change of a dynamic state in a software system. Systems integrated on this level are tightly coupled together. For example, CAM software usually divides its process analysis into numerous steps. On occasion the system may have to roll back a few steps before moving forward further. This event highly suggests the
invalidation of some conclusions from those invalidated steps. Event sharing is very beneficial for those systems which depend on the conclusions to make responses without any delay. The nature of design requires the designing process to move back and forth frequently. Therefore, synchronization is essential for concurrent engineering. Event sharing supports the integrated system for concurrent engineering design.

1.3 Types of Software Integration

An important impetus for the study of software integration comes from the increasingly competitive global market. Easy access to strategic information can provide a competitive advantage and cost savings for manufacturers. Integrating the manufacturing software systems allows the manufacturers to share information between mission-critical and legacy applications located on various computer platforms. There are several different types of software integration found from the literature. These types can be divided into three categories: component-based integration, data model-based integration, and data translation-based integration.

1.3.1 Component Based Integration

The Distributed Component Platforms (DCP)[27] has emerged within the past years and has impacted the software world tremendously. The objective of DCP is to increase the reusability, reliability, and interoperability of software. To achieve this goal, DCP insists that software systems should be developed by integration of reusable building blocks—software components. Each component encapsulates semantically
meaningful application or technical services. The integration of these components should follow component standards which specify how to interconnect them. Components are different from libraries and subroutines. Software components can be customized as binary executable, while the libraries and subroutines must be modified as source code. There are two DCP market leaders at the moment: Microsoft's Distributed Component Object Model (DCOM)[29] and Sun's JavaBeans[40]. DCOM is the extension of Microsoft's Component Object Model (COM)[30]. This component technology is mostly applied to today's PC-based applications. DCOM extended COM so that it is applicable in the distributed environment. DCOM builds an object remote procedure call (ORPC) layer on top of DCE RPC[16] to support remote objects. Enterprise JavaBeans—similar with DCOM to COM—extends Sun's JavaBeans component model to support sever components. It defines a component model for the development of Java applications based on a multitier, distributed object architecture. There is one most distinguishable difference between these two standards. DCOM is language neutral but platform dependent. JavaBeans, in contrast, is language dependent (JAVA) but platform neutral.

The Common Object Request Broker Architecture (CORBA)[41] is a standard specification instead of a specific product of component technology by Object Management Group (OMG), a consortium of more than 700 software vendors, developers, and users. CORBA defines an object oriented approach to create software components that can be reused and shared between applications. In fact, CORBA absorbs a lot of specifications applied by JavaBeans[35][13]. Its core part is Object Request Broker (ORB), which connects the components and makes sure the
components can communicate with one another. CORBA has been thought of as the universal software integration technology over the heterogeneous distributed computing environment of the internet.

One of the areas to which CORBA has been extensively applied is CORBA-based manufacturing system integration. This specialized application is based on reusable dynamic run-time “object oriented” manufacturing system architectures of CORBA[37]. Manufacturing Open Framework Architecture (MOFA)[7] provides an open object-oriented infrastructure which allows applications and components to cooperate and interact through a CORBA compliant interface. It serves as a backbone that provides support for the communication between these individual software elements. MOFA is aimed specifically at manufacturers to provide features demanded by agile and flexible manufacturing. It allows on-line reconfiguration of the application by loading and unloading software “components”. It also provides a very cost-effective way to link old and new applications into a common, synergistic system.

1.3.2 Data Modeling Based Integration

Instead of using interoperable components to fulfill software integration, data modeling based integration applies neutral data representation for inter-system exchange of data. This method is getting more prominent because of the maturity of object oriented technologies[17][15]. Object oriented technologies make it possible and easier to model complex and multi-disciplinary data. Because the data modeling technique is based on an open and extensible set of protocols, the key point of a successful standards is the acceptance by the international community. In fact, this
might be one of the reasons that modeling based integration has been primarily deployed in manufacturing industrial systems. The manufacturing society has developed and adopted several international standards for the exchange of product data. One of the standard is the Standard Exchange Product Data (STEP)[19][28]. STEP is a series of international standards which were prepared by Technical Committee TC184, Industrial automation systems and integration, Subcommittee SC4, Industrial data and global manufacturing languages. Its objective is to “provide a neutral mechanism capable of describing product data throughout the life cycle of a product, independent from any particular system.”[20] Currently the research and application of STEP are very active. For example, STEP AP203 has been implemented to configure controlled three dimensional design[28]. The National Industrial Information Infrastructure Protocol (NIIP)—an open and extensible set of protocols for sharing industrial data over Internet and Intranet[18]—uses the STEP standard to ensure that as much data as possible can be shared.

Enterprise integration is a huge challenge to the manufacturing industry. It needs consensus among all parties involved to reach better integration solutions. CIM Open System Architecture (CIMOSA)[8][23]—the solution for enterprise integration by European Consortium—is another example of model based integration. The ingredients of CIMOSA architecture include reference model and particular model. A reference model “can be considered a general model that can be used as a base to derive other models.” A “particular model is used in a particular enterprise for operation qualification and can be derived from the reference model.”[8] CIMOSA defines four different modeling views: function, information, resource, and organization. In
summary, CIMOSA has the following characteristics:

1. It provides a modeling language sufficient for modeling complex industrial environments.

2. It provides a modeling framework that covers the life cycle of enterprise operation and accommodates different modeling methods.

3. It supports graceful migration through an evolutionary approach of modeling (coexisting with legacy systems).

1.3.3 Data Translation Based Integration

Data translation based integration may or may not depend on the data modeling technique. It does not emphasize a uniform data model among local systems. It tolerates model differences in different systems. The philosophy behind this type of integration is based on effective and efficient data transformation among different data models inside local systems. This type of integration is the basis for the Intelligent Manufacturing Workstation (IMW)[24] and Integrated Manufacturing Design Environment (IMDE)[26].

The IMW is composed of several commercially available tools, including PART—a process planning system, Cost Advantage—a cost evaluation software, and Virtual NC—a 3D simulation software. The input to IMW are STEP files which contain the B-rep description of a part and blank. PART can determine the complete operation plan consisting of setups, machine and cutting tools, and the cutter path described using Automatic Programmed Tools (APT). Then, the cost of operation can be estimated by Cost Advantage. Finally, the kinematics simulation of the whole procedure of this
operation can be simulated by Virtual NC. The simulation can verify the machine program. As shown in Figure 1.1 (adapted from Judd et al, 1994), the architecture of IMW contains the following data translators:

- **PART translator** translates the STEP files of part and blank from CAD tools into ICEM Part.
- **Cost Advantage translator** converts data about the features, machine tool, cutting tools, machine times, and features from PART database into Cost Advantage.
- **Virtual NC translator** extracts the data about machining tool, cutting tools, raw stock, setups, and fixture from an ORACLE database owned by PART into VNC.

The IMDE aims at creating a virtual environment for manufacturing systems design by integrating the existing designing tools. The architecture of IMDE consists of unified data meta-model (UDMM), intelligent interfaces, link manager, message board, and synchronization interfaces and manager. The UDMM defines a standard data format for data translation. However, the UDMM is not the union of data models in the system; instead, it only covers all shared data. Each designing tool has an intelligent interface. The intelligent interface translates data from local models into UDMM format or from DMM format into local format. Its knowledge contains not only its local data model but also the mapping relationship between the local data models and UDMM. The architecture of a typical interface is shown in Figure 1.2 (adopted from Koonce et al, 1996). The link manager links the UDMM and the intelligent interface by developing a set of meta-links between local data model and UDMM. The message board is useful to post unresolvable constraints; therefore, the designer will be able to obtain enough knowledge to get ultimate control. The synchronization interface and manager are
Figure 1.1: Architecture for the Intelligent Machining Workstation
Figure 1.2: Intelligent Interface of IMDE
introduced for dynamic integration, such as the execution of simulation models for the manufacturing systems.

The intelligent interfaces are the key components to the functioning of the IMDE. An intelligent interface can send out data requested by others and receive data requested from others guided by link manager. The incoming data goes through three modules in the following order: analysis, translation, and access to local data. And the outgoing data go through these modules in reverse order. The analysis module is coordinated by the control module, which connects to the link manager, message board, etc. The translation module is controlled by a translation map.

1.4 Existing Tools for Software Integration

Script languages have long served to be languages for software integration. Sometimes a script language is alternatively called a glue language or an integration language[33]. Script languages, as a broad category, have a lot of features which distinguish them from conventional languages, such as C and C++. System programming languages were designed for building data structures and algorithms from scratch. They assume the basis of the computing environment is words of memory. In contrast, script languages are designed for gluing. They assume the existence of a set of powerful components and are intended primarily for connecting components. System programming requires the languages to be strongly typed to manage the complexity of systems. Integration requires the languages to be loosely typed or typeless to ease the connection of components. There are many different script languages in existence today, such as the Unix shell script, Awk, Visual Basic, Tcl, Perl, etc.
1.4.1 Visual Basic

Visual Basic was first designed as system programming language and then changed to become a script language. It is used to develop applications of MS Windows by assembling components which are ActiveX extensions (sometimes called VBX). As discussed previously, DCOM from Microsoft has become one of the leaders of DCP market. Host languages of DCOM can be Visual C++, Visual J++, and Visual Basic.

1.4.2 Awk

For years, Awk has been proven to be a robust yet simple language for scanning text files. Its “basic operation is to search a set of files for patterns, and to perform specified actions upon lines or fields of lines which contain instances of those patterns.”[3] Similar to another UNIX utility grep, Awk scans an input file record-by-record and splits each record into fields. Both the record separator and field delimiter can be reset with default values: a newline for record separator and white space (blanks or tabs) for field delimiter.

The robustness of Awk mostly relies on its scanning and acting abilities. Patterns in Awk can be very complex because it supports the combination of regular expressions. The action in Awk may consist of arithmetic and string expressions and assignments, looping statements, and multiple output streams.

The simplicity of Awk results from the same features that many other script languages possess. Awk integrates strings and numbers completely; i.e., values in Awk can be both strings and numbers. Awk determines appropriate value types until the last
moment of their use. This feature in some degree decreases the speed of the program, but it makes development much easier.

1.4.3 Tcl

Tcl as “tool command language”[32] is one of the two software packages (the other part is called Tk) freely available on the Internet. Together these two packages “provide a programming system for developing and using graphical user interface (GUI) application.”[32] Tcl was designed for controlling and extending applications with generic programming facilities, such as variables, loops, and procedures. One of the advantages of Tcl compared with other script languages is its extensibility. Its interpreter consists only of a library of C procedures so that it is easy for an application to have its own powerful scripting language based on Tcl.

Tcl scripts usually are used as a communication mechanism to allow different applications to work together. For example, any windowing application based on Tk can send a Tcl script to any other Tk application to be executed there. In spite of its power, Tcl is rather simple. It is suitable for rapid development.

Currently, there are a lot of extensions of Tcl/Tk in its community. These extensions not only extend the application domain of Tcl/Tk, but also prove Tcl/Tk's popularity.

1.4.4 Perl

Perl—abbreviated from Practical Extraction and Report Language or Pathologically Eclectic Rubbish Lister[42]—is a script language for easily manipulating
text, files, and processes. In the beginning, Perl was intended to be a data reduction language—a language for navigating among various files in an arbitrary fashion, scanning large amounts of text efficiently, invoking commands to obtain dynamic data, and printing easily formatted reports based on the information gleaned[42]. Later, Perl became a language which can also conveniently manipulate files and processes. It is capable of creating, removing, renaming, and changing permission of files. It is also capable of creating and destroying processes and dealing with interprocess communication. To summarize, Perl has the following features:

1. Perl was derived in spirit from other portion of UNIX (although it runs on other platforms, too). It was structured very similar to C language.

2. Perl has rich syntax and many unconventional symbols—operators which cannot be found in other languages.

3. Perl eliminates a lot of the limitations of UNIX utilities, such as length of file names, variable names, and so on.

4. Perl supports high performance associative arrays implemented with hash tables.

5. Perl provides data flow tracing mechanism to determine which data is derived from insecure sources. Therefore, Perl program is more secure.

1.5 Problem Statement

The integration using data translators has been proven to be a prominent method. The ingredients of a data translator can be divided into three aspects: data reading/writing, data analysis, and event controlling.
• **Data reading/writing** is quite challenging for data translators. Software systems have many and various data storage methods: ASCII files, relational databases (e.g. ORACLE database), STEP databases, etc. In addition, data may also come from the machine's standard input or from networks. The extensive data communication requires a data translator to have an easy and efficient facility for both reading and writing data.

• **Data analysis** includes both mathematical analysis and string analysis. Mathematical analysis encompasses numerical value comparison and calculation. String analysis consists of string comparison and string manipulations.

• **Event controlling** is required sometimes by a data translator to issue commands to get some dynamic data from a system.

The benefits of this method can be summarized as follows. First, data translation method eliminates the requirements of any standards. Component based integration and data modeling based methods both mainly require that systems are prepared for integration, both of them must carry mechanisms for dealing with non-standard components (CORBA) or models (CIMOSA). Therefore, the data translation method is especially suitable for legacy systems. Second, the data translation method is rather simple and straightforward compared with component platform technology. It does not require an understanding of all the component interfaces, which in most of cases can be very complicated. This is particularly attractive for rapid prototyping of integrated systems. Despite its attractive side, a data translator method still faces some challenges. The first challenge is that this method depends on how open the software systems are. In other words, it can only translate those explicitly stored data of a system.
Accessibility to other data in the system depends much on the APIs the system provides. Fortunately this is not a serious problem when dealing with well-designed software systems.

The second challenge is its implementation. To date, there is no easy way to implement a data translator. For example, the IMW[24] required more than eighteen months for its implementation; and the code of the translators in IMW were the mixture of Awk, UNIX shell scripts, and C programs. The complex technology translators, such as the STEP to VRML and STEP to CAD translators, require a large time investment to implement (greater than six person months)[18]. This challenge actually is inherited from the core of software integration: to deal with the data of heterogeneous systems. Existing integration tools have strong points on data analysis and data manipulation. However, they are short on abilities to deal with many and various data accessing requirements, which are needed to connect to a relational database server and extract data from it. The existing choice is using SQL and a host procedural language like C; however, a system language like C is not easy to use for data analysis and data manipulation compared with script languages. Therefore, a data translator code may contain a mixture of C (plus SQL) and script languages (e.g. Awk, Unix shell scripts). The mixture codes are not only hard to develop but also hard to maintain. Another implementation difficulty comes from the fact that there is yet no easy way to access STEP data. STEP data has been a choice of neutral data format because of its robustness and international recognition. The UDMM in IMDE[26] applies STEP format even if it is not required. The National Industrial Information Infrastructure Protocols (NIIIP)[18] adopted the STEP standard. While the STEP standard has been accepted to be a choice
of data integration, there is no high level language that can be used to access and manipulate STEP data. So far, the access of data in STEP must be limited to using Standard Data Access Interface (SDAI)[21], a part of STEP series standard (e.g. SDAI C binding (Part 24)).

To make the implementation of the data translator easier, firstly, a higher level query language on STEP data is needed. Secondly, an integrated programming environment is needed. This integrated programming system makes the implementation of data translators not only in a unified form, but also makes the code more efficient and maintainable.

### 1.6 Proposal: EXPRESS Query Language and Templates and Rules—Languages for Software Integration

An integrated programming environment consisting of two stand-alone language tools is proposed. Although these two languages can be used separately to perform other programming tasks because of their general purposes, working together these two tools provides an integrated and efficient way to develop data translators. The first language is called EXPRESS Query language (Eql). This language provides a high level query language to specify and perform queries on STEP data. The language will be built on SDAI specifications, therefore, it will work with any data in STEP format. The second language is called Templates and Rules (Tnr) which stands for Template and Rule and supports the unified interfaces to different data resources and powerful data manipulation capabilities. Eql and Tnr are briefly outlined in the next two sections.
1.6.1 EXPRESS Query Language

To date, the access to STEP files has been limited to Standard Data Access Interface (SDAI), a programming interface to EXPRESS defined data. This limitation makes the common query and data manipulation on a STEP file very tedious. A simple query on the contents of a STEP file takes between ten to twenty lines of C code and requires the programmer's acquaintance of SDAI library. The proposed high level query language is going to be provide a powerful tool to make query statements that can be used to create, remove, and update information of a STEP file based database.

The schema of a STEP file is described by EXPRESS[22] data definition language. EXPRESS is an object oriented data definition language. It provides syntax for definition of classes (entities) and relationships among classes. The cardinality of a relationship can be one to one, one to many, or many to many. The inheritance can be of AND, OR, or AND/OR three different types. Unlike relational query languages which have a relational completeness definition based on relational algebra[14], object oriented query languages so far, have no completeness definition proposed [10]. Eql learns some syntax from other existing query languages both for relational and object oriented data bases, such as SQL, OPM[12], and O2. The syntax of Eql should be tightly related to EXPRESS, the object oriented data definition language which STEP files conform. The goal of this language is to make the query on STEP files easy to state and to perform. This aspect can be embodied by the following guidelines for Eql.

1. Since EXPRESS schema make it possible to define inverse relationships between attributes for two entities, Eql should provides syntax and mechanisms to facilitate queries on this types of relationships.
2. EXPRESS defines a complete set of aggregation data types: list, array, set, and bag. Eql, therefore, should provide aggregation operators and statements. For example, an operator for breaking an aggregate into elements and a statement for traverse through an aggregate are helpful.

3. Eql should directly access or manipulate attribute values of instances in its queries (unlike O2, which does this via methods of classes). The reason for this difference is that EXPRESS does not provides methods to manipulate attributes (all attributes are public). Therefore data integration is a potential problem of STEP files. EXPRESS deals with this problem by constraints of local and global rules.

4. Eql must provide other system levels commands in addition of the general purposes query statements. These commands, such as creating STEP files, opening a STEP file, etc. should be provided to Eql complete.

1.6.2 Templates and Rules

One of the challenges of implementation of data translators is how to extract data from many and various resources: text files, relational databases, STEP databases, network, and so on. System languages such as C are capable but too tedious for such common tasks. No single existing script language directly supports access to all these resources even though they provide some powerful features such as data analysis and scanning text files etc. The proposed language Templates and Rules (Tnr) aims to be an integrated programming system which supports all the different types of data access. At the same time, Tnr is going to inherit some simple syntax from other languages to deal with data analysis.
Figure 1.3[25] shows a typical environment of a data translator implemented by Tnr. The features of Tnr can be outlined as below.

1. Tnr should provide facilities to communicate various kinds of information resources including: relational databases, STEP files, ASCII files, and networks. Communication with relational databases such as Oracle, Informix, etc. can be done via SQL statements. Tnr should have functions to send SQL statements to a database server and receive the query results from the server. Communication with a STEP file can be achieved by sending Eql Statements to the Eql query engine and receive query results from it. Scanning text files has a lot of examples from other script language, such as Awk, Perl etc. Communication via network can be achieved by UNIX socket.

2. A pattern is a set of objects with some recognizable property. One type of pattern is a set of character strings[5]. Pattern recognition is critical for a information processing tool and this ability can be achieved by using regular expressions. Tnr provides the user plenty of functions related to regular expressions. We can use these functions search through ASCII text files, standard input, a string in temporary memory, etc.

3. Information processing can be divided into two activities: scanning information resources and generating new format of information. The separation of these two activities can be seen in some scripting languages. For example, the program structure for Awk contains a sequence of statements of the form

    pattern {action}
    pattern {action}
    ...


Figure 1.3: A typical data translator implemented with Tnr
4. Each record of input is matched against each of the pattern in turn and the associated action is performed upon the pattern is satisfied. Tnr should have some similar feature that separates the two activities. It is proposed to support two different types of procedures: rules and templates. A rule procedure is read only procedure. Its purposes are to extract information from files while the template is to generate new files containing the new format of data.

5. Variables in scripting languages usually do not have data types. This feature doesn't happen by accident. Gluing languages integrate strings and numbers and try to make decisions on type as late as possible. Tnr deals with variables by the same way. As an example, Tnr knows a plus operator expects its two operands to be two numbers. Therefore, it changes these two operands to numerical values if they are not. Some scripting languages do not need to declare variables. This is not true for Tnr. No declaration may be a good idea for a simple programming task. But Tnr is designed for some more complex computing tasks.

6. “Garbage collection” technology has been existing for years. It was first found in scripting languages such as LISP. However, it can be found even in today's system languages. In Java, memory sections without any references pointed to are released automatically. Garbage collection dramatically reduces the effort needed for memory management. Therefore it allows a programmer to concentrate on the implementation of the algorithm itself. In Tnr, the array size is automatically increased upon need and the memory allocated for local variables are automatically deallocated when the execution goes beyond the function.
7. Tnr does not support user defined data structures. Whenever a new data structure is necessary, a user defined EXPRESS schema can be deployed and a STEP file can be applied to serve as temporary data depository. However, Tnr supports a built-in data structure: array. The array-based representation of lists is in many ways more convenient than the linked-list representation. Tnr makes it more convenient by removing the limitation of the length of arrays. Arrays have a lot of applications in data processing, for example, arrays are suitable to store clusters of information returned from relational database servers or Eq1 query engines; a string can be split based of the specified field separator into an array; elements inside an array can be merged together.

8. Tnr supports functional programming. A Tnr program always consists of procedures with a minimum program containing only one procedure called main. Two fundamental procedural types (rule and template) are supported by Tnr to facilitate two major kinds of jobs (input to and output from a Tnr program). The main procedure can be of either type. A procedure can return to its caller a scalar value. A rule procedure also uses a matched or failed statement to return. Calling a procedure can be done by two means. If the caller only wants to bring a value into the callee via a parameter, call-by-value can be applied; if the caller wants to bring a value out from the callee, call-by-reference can be applied. In the later case, the caller places a & before the variable argument. Remember that the callee can return to the caller a scalar value using a return statement, but if an array value is what is intended to be returned, call-by-reference is the only means to do it. Parameters to a procedure have local scope. Recursion is also supported by Tnr.
In summary, Tnr is a novel programming language. Its syntax is a mixture of several familiar languages like C, C++, Awk, and LISP. Its performance certainly slows down compared with system languages like C. But its speed should still be workable, especially because of the hardware improvements in recent years. The easy syntax and direct support for access to multiple information resources are the two most important characteristics for Tnr.

1.7 Conclusion

One of the prominent methods for software integration is to use data translators. Data translators perform the following tasks: data extraction, data analysis, and data generation. Due to the heterogeneous characteristics of software systems, data translators are hard to implemented currently. This difficulty can be overcome by inventing a new computer language to directly support the computing that a data translator requires. Such a language should be able to directly access text files, relational databases, STEP files, networks, etc. It should also support data analysis facilities such as arithmetical and string operations. In addition such a language needs to be capable of process controlling etc. This research is going to present two languages Eql and Tnr to fulfill these requirements.
Chapter 2

The EXPRESS Query Language

2.1 Introduction

The STEP exchange file standard provides not only the neutral data format to exchange product data among different software systems, but also the means to construct the common data repositories or databases. Like other object oriented data modeling schemas, EXPRESS describes the real world (product) using objects, each of which has a unique I.D. It also supports inheritance relationships among entities on the basis of their common properties. All other arbitrary kinds of relationships can be qualified by attributes.

Currently, manipulation of a STEP exchange file is limited to using PART 22 (SDAI). SDAI defines a set of low-level interfaces to STEP data, but these interfaces are too tedious to support large-scale data access and manipulation. Therefore, a high level access utility is needed, and this utility can and should be built on the top of the low-level SDAI interfaces.

In this chapter a novel query language called EXPRESS Query Language (Eql) is introduced. Eql is similar in structure to the SQL-like query languages supported by
object-oriented DBMS, such as O2[9] or other types of hybrid query languages, which are based on object-oriented data modeling methods, such as OPM[12]. Eql queries can be categorized into five types: select, update, insert, delete, and system-level commands. Select queries are used to extract data out of a database according to particular conditions. Update queries modify attribute values of existing objects in a database. Insert queries are used to create new objects in a database while delete queries are used to remove objects from a database. System level commands, such as open, close, etc., manage a database on system levels.

First in this chapter, the background of STEP (ISO 10303) is reviewed because of its direct relationship with Eql. Then the syntax of Eql will be detailed.

2.2 ISO 10303

ISO 10303, a series of international standards, were prepared by Technical Committee TC184, industrial automation systems and integration, Subcommittee SC4, industrial data and global manufacturing languages. Its objective is to “provide a neutral mechanism capable of describing product data throughout the life cycle of a product, independent from any particular system.”[20] “The nature of this description makes it suitable not only for neutral file exchange, but also as a basis for implementing and sharing product databases and archiving.”[20] The standards are more commonly referred as the Standard for the Exchange of Product Model Data (STEP). It largely consists of the following sub standards[20]:

1. Part 1 describes the overview and fundamental principles;
2. Part 11 specifies the description method;
3. Parts 21 and 22 specify the implementation methods;
4. Parts 31 and 32 specify the conformance testing methodology and framework;
5. Parts 41 to 49 specify the integrated generic resources;
6. Parts 101 to 105 specify the integrated application resources; and
7. Parts 201 to 213 specify the application protocols.

Parts 11, 21, and 22 are briefly reviewed in the following three subsections as they are of direct interest to Eql.

2.2.1 ISO 10303-11

The name for this standard is *Description method: The EXPRESS language manual*. It contains two major parts. The first is an object-oriented data definition language called EXPRESS, whose objective is to specify the aspects of product data. The second part is a graphical representation specification called EXPRESS-G, which represents subsets of the constructs in the language. Before an exchange file can be generated, one should first create a schema in EXPRESS for it. The schema specifies both the definitions of data types and the constraints on their instances. EXPRESS supports the following data types:

- **Simple data types** are a group of atomic data types, including real, integer, number, string, boolean, logical, and binary. The domain of real covers all rational, irrational, and scientific real numbers. Integer is the domain for all integer numbers. The domain of number is the union of real and integer. String data type has the domain of a sequence of characters, which are defined in ISO 10646. Boolean only contains two literals—true and false—while domain of logical is the superset of that of
boolean. It has three literals—true, false and unknown. Binary data type has its domain as sequences of bits (0 or 1).

- **Aggregate data types** supported by EXPRESS include array, list, set, and bag.

  1. **Array** is an indexed, fixed-size collections of elements. The elements must be of the same data type. The index should be integer and can start from any number. The element value is not necessarily unique; even the user can optionally specify so.

  2. **List** is an ordered sequence of elements. It specifies the minimum and maximum number of elements that can be held by optional lower and upper bounds. It also can specify the uniqueness of elements.

  3. **Set** is the domain of an unordered, non-duplicated collection of like elements. The optional lower and upper bounds specify the minimum and maximum number of elements it can hold.

  4. **Bag** defines the domain of an unordered collection of like elements. It also has optional lower and upper bounds too. Duplicated elements are permitted.

- **Named data types** are the data types that may be declared in a formal specification. There are two kinds of named data types: entity data type and defined data type. An entity data type is assigned and referenced by an entity identifier and a defined data type by a type identifier. A defined data type is simply an alias of an EXPRESS built-in atomic data type or a constructed data type (see below), while an entity data type is a user defined type. An entity data type contains a group of attributes to describe the objects in the real world.
- **Constructed data types** can be either of enumeration or select data type. They cannot be directly used as base data type; namely, they cannot be used to represent attribute data type or aggregate element data type. If this is the case, one should use named data types.

1. **Enumeration data type** has its domain as an ordered set of names. The names represent values of the enumeration data type.

2. **Select data type** has its domain as the union of the domain of the named data type in its select list.

- **Generalized data type** is the last of several groups of EXPRESS data types. It is used to specify a generalization of certain other data types and only is useful in the definition of functions and procedures.

### 2.2.2 ISO 10303-21

ISO 10303-21, named as *Implementation methods: Clear text encoding of the exchange structure*, is the rules for creating a STEP file based on an EXPRESS schema. Clear text encoding encodes information using 8-bit byte values corresponding to the basic alphabet. Part 21 requires an exchange structure to contain two sections: a header section and a data section. The header section should contain at least three pre-defined entity instances: a file_description instance, a file_name instance, and a file_schema instance. It also permits a few user defined entity instances in this section. A data section holds all of the user's data, each of which is an entity instance. The entity definitions for all these instances should be defined in the schema, whose name is
specified in the file_schema instance in the header section. A user instance is identified by its unique name, whose format consists of # and a positive integer number.

An optional value, expressed as $ in an exchange structure, indicates that the value for an optional attribute or for an optional element of an array is not provided.

2.2.3 ISO 10303-22

Part 22 defines the standard data access interface (SDAI) to a STEP exchange structure. There are two different SDAI binding systems—"early-bind" (e.g., SDAI C++, SDAI Java, and SDAI CORBA/IDL) and "late-bind" (e.g., SDAI C). The early binding system creates specific programming language data structure for each definition in an EXPRESS schema while the late binding system uses an EXPRESS data dictionary and a set of normalized functions to access data values. Apparently, implementation of Eql should apply a late binding system as there is no way to anticipate the EXPRESS schema at the time of Eql implementation. The various types of binding systems are commercially available, such as ST-Developer from STEP Tools, Inc.

2.3 Examples Used In This Chapter

This section presents an example EXPRESS schema which may be oversimplified for practical use. The purpose of this example and an associated example STEP database are for illustrating concepts, syntax, and other fine points of Eql in the latter sections in this chapter. The EXPRESS schema called "university" is given below.
SCHEMA UNIVERSITY;

TYPE QUARTER = ENUMERATION OF (FALL, WINTER, SPRING, SUMMER);
END_TYPE;

ENTITY CLASS_OFFERING;
  title: STRING;
  id: STRING;
  callNumber: STRING;
  theQuarter: QUARTER;
  year: INTEGER;
  theInstructor: INSTRUCTOR;
  theAssistant: OPTIONAL ASSISTANT;
  gradeMap: ARRAY [0:11] OF INTEGER;
  enrollments: SET OF ENROLLMENT;
  assignments: OPTIONAL SET OF ASSIGNMENT;
  exams: OPTIONAL SET OF EXAM;
  assignmentWeight: REAL;
  examWeight: REAL;
UNIQUE key: id;
WHERE sumOfWeight: assignmentWeight + examWeight = 1.0;
END_ENTITY;

ENTITY PEOPLE ABSTRACT SUPERTYPE OF (ONEOF (INSTRUCTOR, STUDENT));
  firstName: STRING;
  middleName: OPTIONAL STRING;
  lastName: STRING;
  id: STRING;
UNIQUE key: id;
END_ENTITY;

ENTITY INSTRUCTOR SUBTYPE OF (PEOPLE);
  classes: SET OF CLASS_OFFERING;
END_ENTITY;
ENTITY STUDENT SUBTYPE OF (PEOPLE);
enrollments: SET OF ENROLLMENT;
END_ENTITY;

ENTITY ASSISTANT SUBTYPE OF (STUDENT);
classes: SET OF CLASS_OFFERING;
END_ENTITY;

ENTITY ENROLLMENT;
  theStudent: STUDENT;
  theClass: CLASS_OFFERING;
  courseGrade: OPTIONAL REAL;
  assignmentGrades: OPTIONAL SET OF GRADE;
  examGrades: OPTIONAL SET OF GRADE;
UNIQUE
  key: theStudent, theClass;
END_ENTITY;

ENTITY GRADE_WORK ABSTRACT SUPERTYPE OF (ONEOF (ASSIGNMENT, EXAM));
  name: STRING;
  description: STRING;
  possibleScore: REAL;
  weight: REAL;
  theClass: CLASS_OFFERING;
  (* grades: SET OF GRADE; *)
UNIQUE
  key: name, theClass;
END_ENTITY;

ENTITY ASSIGNMENT SUBTYPE OF (GRADE_WORK);
END_ENTITY;

ENTITY EXAM SUBTYPE OF (GRADE_WORK);
END_ENTITY;

ENTITY GRADE;
  grade: REAL;
  (* theStudent: STUDENT; *)
  theWork: GRADE_WORK;
END_ENTITY;

END_SCHEMA;
A university database is used to keep track of the class offerings and their enrollments for each quarter at a university. A class has a call number, a title, and an I.D. It must be taught by one instructor and an optional teaching assistant. A teaching assistant should be a student who is not taking that class. For each class, all the information regarding homework, projects, and exams should be recorded, including weights for the descriptions, final scores, and maximum scores. Other reasonable information regarding faculties and students, such as names, social security numbers, etc., should also be stored in the database. The example STEP file is given below.

ISO-10303-21;
HEADER;
/* Exchange file generated using ST-DEVELOPER v1.5 */

FILE_DESCRIPTION(
/* description */ '(',
/* implementation_level */ '/2;1');

FILE_NAME(
/* name */ 'hw9',
/* time_stamp */ '/98-04-08T11:44:49-04:00',
/* author */ (''),
/* organization */ (''),
/* preprocessor_version */ '/ST-DEVELOPER v1.5',
/* originating_system */ '..',
/* authorisation */ '/');

FILE_SCHEMA ('university');
ENDSEC;

DATA;
#1=CLASS_OFFERING('Graduate Coloquim','12345','EE690',WINTER,1998,
$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,$,
#10=CLASS_OFFERING('Advanced CIM','12345','EE690',WINTER.,1998,
#60,0,
(60,65,70,75,80,83,85,87,90,92,94),
(#660,#860),
(#100,#110,#120),
(#360,#370,#380),
5.0E-01,5.0E-01);
#20=CLASS_OFFERING('Programming With C++','67890','EE590',FALL.,1997,
#60,0,
(60,65,70,75,80,83,85,87,90,92,94),
(#1000,#1130),
(#160,#170,#180),
(#360,#370),
6.0E-01,4.0E-01);
#30=CLASS_OFFERING('Statistics','23456','ISE490',WINTER.,1998,
#70,0,
(60,65,70,75,80,83,85,87,90,92,94),
(#1320,#1380),
(#260,#270,#280),
(#410,#420),
4.0E-01,6.0E-01);
#40=CLASS_OFFERING('Data Modeling','34567','ISE700',SPRING.,1999,
#70,0,
(60,65,70,75,80,83,85,87,90,92,94),
(#1440),
(#290,#310),
(#430,#440),
3.0E-01,7.0E-01);
#50=CLASS_OFFERING('Dissertation','45678','ET855',SUMMER.,1998,
#80,0,
(60,65,70,75,80,83,85,87,90,92,94),
(#1760),
(),
(),
$,$);

#60=INSTRUCTOR('Bob','P','Judd','1347832134',(#10,#20));
#70=INSTRUCTOR('David','A','Koonce','4758943275',(#30,#40));
#80=INSTRUCTOR('Charles','M','Parks','8432967583',(#50));

#100=ASSIGNMENT('hw1','easy',1.0E+01,1.0E+00,#10);
#110=ASSIGNMENT('hw2','moderate',1.0E+01,1.0E+00,#10);
#120=ASSIGNMENT('hw3','difficult',2.0E+01,1.0E+00,#10);
#160=ASSIGNMENT('hw1','easy',1.0E+01,1.0E+00,#20);
2.4 Overview and Glossaries

Eql queries are very structured. They are constructed by one or more clauses, each of which optionally consists of keywords, constants, classes, attributes, attribute expressions, and conditional operators. In this section, a list of Eql building blocks are introduced.

- **Entity** is represented by its corresponding name string in an Eql query. It stands for the set of all the instances of its kind (including its direct and indirect descendants)
in a model. Internally it is described by an array of instance I.D.s. For example, entity *People* represents a set which contains all faculties, students, and teaching assistants, i.e., (#60, #70, #80, #90, #500, #510, #520, #530, #540, #550, #560).¹

- **Attribute** is represented by its name string in Eql. It is only meaningful when it is associated with a specific instance. It stands for the corresponding attribute value of that instance, which can be either scalar or aggregate. Attributes in EXPRESS can be divided into three groups: property attribute, derived attribute, and inverse attribute. A derived attribute can be expressed by one or more property (normal) attributes. An inverse attribute is used to constrain the cardinality of the relationship between two classes.

- **Attribute Expression** is a path of attributes from a parent class to its direct or indirect descendants classes separated by delimiter “.” (“->” is used between a variable and an attribute). A path should start with an instance equivalent expression (e.g. an entity, or an instance variable). The data type of an attribute expression is of that of the last attribute in the path. It is obvious that the data types of the middle attributes inside a path expression must be of entity data type or aggregate data type whose eventual base type is of entity data type. In the latter case, the attribute of aggregate data type should be expanded into elements using one or more [I] operators. Because an entity name always stands for a set of instances in an attribute expression, it becomes too tedious to use this operator after each entity name. Therefore, Eql implicitly appends a [I] operator after an entity name. It is an

¹Even EXPRESS database assigns each object a persistent I.D., SDAI does not require the availability of this I.D. Therefore, the way of constructing I.D.s is system-dependent
error if the user puts [] after an entity name because an array of instances cannot be broken twice. The last special operator is the index operator and represented by notation [*]. This operator applies an aggregate expression and returns an array of indices of the elements in expression. For example, “X[*]” returns (1, 2, ..., 11) if X stands for all people and is equal to (#60, #70, #80, #90, #500, #510, #520, #530, #540, #550, #560).

- **Variable** is a user-defined name string that stands for a single or a collection of data. The name of the string should start with a letter or an underscore character ‘_’, followed by zero or more letters, underscores, or digits. Furthermore, a name should not be equal to any keywords. There are two places a variable can be declared. One place is in from-clauses; the second place is in for-condition in WHERE clauses. In the former case, the variable is visible to the whole query; while in the latter case, the variable is only valid within its local condition. We will discuss the usage of variables in more detail in Sections 2.5 and 2.6.8.

- **Constants** can be of either scalar or aggregate. Scalars include integer, real, string, enumeration, boolean, logical, and instance. Aggregate constants can have arbitrary depths, but the level of each element in a nest aggregate must be constant (the depth of $\$ \text{varies depending on where it is used}$). The notations for all different types of constants conform to the Part 21 standard. For example, a string should be expressed by an array of characters surrounded by single quotes. An aggregate can be expressed as a list of elements separated by commas and surrounded by parentheses. A handy application of aggregate constants can be found in an insert

and these I.D.s are not necessarily persistent.
query and update query, where an aggregate value may be used to initialize or update aggregate attributes of a new instance. In this case, an aggregate constant can be interpreted as array, list, set, or bag, depending on the attribute data type. The formal definition of a constant is illustrated in Figure 2.1.

```
ValueList       : Value
| ValueList ',' Value
;
Value           : PrimitiveValue
| Aggregate      
;
PrimitiveValue  : INTEGER_TOKEN
| REAL_TOKEN
| STRING_TOKEN
| '#{D+}'
| '.[uU]'`
| '.[tT]'`
| '.[fF]'`
| EnumerationValue
| '$'
| HostVariable   
;
EnumerationValue: '.' NAME_TOKEN '.'
;
Aggregate       : '(' ')
| '(' ValueList ')' 
;
```

Figure 2.1: Grammar rules for constant values

### 2.5 FROM Clause

In SQL, a from-clause specifies the working relations (tables) in database; when two relations are needed (e.g. join of two tables) an alias of tables can be possibly assigned. In Eql, however, from-clauses have much different purposes. First, they collect data from the database (a data model) and associate the data with variables. For
example, statement “FROM X IN STUDENT” searches the whole working data model and collects all instances of students and associates the data with variable X. Therefore, $X = (\#500, \#510, \#520, \#530, \#540, \#550, \#560)$. The second purpose of a from clause is to alias an expression. In this case, a declaration of a variable does not collect any data from the model; it simply makes a symbol for an expression. Afterwards, everywhere the expression is expected, the symbol can and should be used. Sometimes a query can become rather simple when it deals with only a single class. In this case, a from-clause may use an implicit variable instead of an explicit one. We call this type of from clause “implied from clause”. The normative grammar definition for a from statement is shown in Figure 2.2. Notice that it is also allowed to declare a variable to be associated with a constant. For example “FROM X IN (1, 2, 3)” is a valid statement. Statements of this type are usually used in for-condition to traverse an aggregate (see Section 2.6.8).
Figure 2.2: Grammar rules for the FROM-statement

### 2.6 WHERE Clause

As we have discussed, the from-clause in a query collects raw data from a model. In most of cases, not all of these collected data are subject to further actions, i.e., select, update, or delete. Therefore a “filter” is needed to filter out the unwanted data collected by the from-clause. This filter is specified by a where-clause, which is constructed by one or more conditions. From this point of view, a where-clause of Eql is similar to that in SQL. The conditions in a where-clause can be divided into atomic and compound conditions. Atomic conditions include NULL condition, EMPTY condition, ISA condition, IN condition, CONTAINS condition, and FOR condition. A compound condition is a combination of two or more atomic conditions combined by logical operators AND, OR, and NOT. Most of the atomic conditions (except
comparison conditions) have two versions—one is positive and the other one is negative (see syntax definition in Figure 2.3). The negative condition, under most circumstances, equals the reverse result of the positive condition. The only exception is when one or two operands in a condition (except for NULL condition) is $\$; both versions return false.

\[
\text{Condition} : \text{Condition AND Condition} \\
| \text{Condition OR Condition} \\
| \text{AtomicCondition} \\
| \text{NOT Condition} \\
| '(' \text{Condition } ')'
\]

\[
\text{AtomicCondition} : \text{NullCondition} \\
| \text{ISACondition} \\
| \text{EmptyCondition} \\
| \text{INCondition} \\
| \text{ContainsCondition} \\
| \text{MatchCondition} \\
| \text{ComparisonCondition} \\
| \text{ForCondition}
\]

Figure 2.3: The syntax definition for conditions

2.6.1 NULL Condition

Null condition (Figure 2.4) makes judgments on whether an expression's value is known or not. Not all instances of an entity have values for all attributes, and an instance does not always bear values for all attributes during its lifetime.
NullCondition : Operand IS NULL

| Operand IS NOT NULL

Figure 2.4: The syntax definition for NULL conditions

For example, a user may want to create a class_offering instance in the database before the instructor for that class is decided (therefore, the value for attribute instructor is unknown). STEP copes with this situation by providing a special symbol `$` for an unknown value. By definition, in a STEP file only an optional attribute value or an element in an array can be unknown. An unknown value is not equivalent to an empty string or a zero integer or an empty aggregate value. The latter have clearly defined specific values. Null value is propagatable; namely, when one attribute value is `$` in an attribute expression path, the value of the expression is `$`. For example, let X be a class_offering instance variable whose I.D. is #1; then expression “X->theInstructor.classes” (means the classes taught by the instructor who teaches the class with I.D. #1) is `$`; and the condition “X->theInstructor.classes is null” is true. The semantics of NOT NULL condition is exactly reversed from NULL condition.

2.6.2 EMPTY Condition

Empty condition (Figure 2.5) tests against an aggregate expression to see whether its value is empty (no elements contained). When this condition is used to test a scalar-value expression, a runtime error arises. Be cautious that an aggregate containing unknown elements (e.g. `($)`) itself is not empty. `$` is neither empty nor not empty.
EmptyCondition : AggrOperand IS EMPTY
| AggrOperand IS NOT EMPTY
;

Figure 2.5: The syntax definition for EMPTY conditions

2.6.3 ISA Condition

When an expression is an entity instance type, one may want to test its belongingness (entity name). Eql supplies an ISA condition (Figure 2.6) for this type of testing requirement. The left operand should be an instance or $; the right operand should be a name token (without single quotes) for an entity name. If the entity name for the instance matches the name token or any of its children's, then ISA condition returns true. Otherwise it returns false.

ISACondition : Operand IS A NAME_TOKEN
| Operand IS NOT A TOKEN NAME_TOKEN
;

Figure 2.6: The syntax definition for ISA conditions

2.6.4 Comparison Condition

Comparison conditions (Figure 2.7) include a total of six types: equal(==), not equal (!=), less than(<), greater than(>), less equal(<=), and greater equal(>=). Different data types are generally not comparable. The only exception is when two data types can be converted meaningfully. For example, a boolean value can meaningfully
be compared with a logical value. Table 2.1 details the comparable data types and the comparison operators.

Table 2.1: Comparable Data Type (*: for == and != only; ** for All Comparison Operators; Otherwise, No Comparison Defined)

<table>
<thead>
<tr>
<th></th>
<th>bi</th>
<th>bool</th>
<th>enum</th>
<th>inst</th>
<th>int</th>
<th>logic</th>
<th>real</th>
<th>str</th>
<th>array</th>
<th>list</th>
<th>set</th>
<th>bag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bool</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Int</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Str</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>List</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ComparisonCondition : Operand `==` Operand

| Operand `!=' Operand
| Operand `<` Operand
| Operand `>' Operand
| Operand `=' Operand
| AggrOperand `==` AggrOperand
| AggrOperand `!=' AggrOperand
| AggrOperand `<` AggrOperand
| AggrOperand `>' AggrOperand
| AggrOperand `=' AggrOperand
| AggrOperand `>=` AggrOperand
| AggrOperand `=' AggrOperand

Figure 2.7: The syntax definition for comparison conditions
The rules for aggregate comparison are stated below. For list and array, only `==` and `!=` operator can be used. If both lists or arrays have the same sizes and all pairs of elements in order are equal, then they are equal; otherwise, they are not. For a set, all comparison operators are supported. If $A \cap B = A \cup B$ where $A$ and $B$ are two sets, then $A == B$, $A >= B$, and $B <= A$ return true; if $A \subseteq B$ then $A < B$, $A <= B$, and $A != B$ return true; for all other cases, except $A != B$, the comparisons return false. Comparisons between bag values are similar to set except that for those with repeating elements, the equal operator also requires the count numbers in both values to match.

For the nested aggregates, the above rules can be applied recursively. The comparison between an enumeration value and a string value first converts the name of the enumeration into a string and then compares both operands as strings. For example, `.black. == 'black'` returns true. Other comparisons are straightforward.

2.6.5 IN Condition

Using the IN condition (Figure 2.8), one can conveniently check the membership of an aggregate value. The aggregate level of the left operand must be one less than that for the right operand (aggregate level for scalar is zero). If the left operand can be found in the next level element list of the right operand, then this condition returns true. If either operand is $\$, then it returns false. All the comparison rules follow those for `==` operator in comparison condition (see Section 2.6.4). Any violations of these rules will cause run time errors.
INCondition : Operand IN AggrOperand
| AggrOperand IN AggrOperand
| Operand NOT IN AggrOperand
| AggrOperand NOT IN AggrOperand

Figure 2.8: The syntax definition for IN conditions

2.6.6 CONTAINS Condition

Contains condition (Figure 2.9) compares two aggregates. Both operands should have the same level and the level must be greater than or equal to one; otherwise, it reports an error. If all the elements in the right operand can be found in the left operand and in case that right operand has repeating elements and the left operand contains at least that many elements, then this condition returns true. If either operand is $, then it returns false. The comparisons of elements follow the rules for == operator in comparison condition. (Refer to Section 2.6.4 for more details.)

ContainsCondition : AggrOperand CONTAINS AggrOperand
| AggrOperand NOT CONTAINS AggrOperand

Figure 2.9: The syntax definition for CONTAINS conditions

2.6.7 MATCH Condition

The MATCH condition (Figure 2.10) is similar to the LIKE condition in SQL. However, it provides more power to recognize character patterns by using regular
expressions. The MATCH checks the string value against a regular expression. The usage of regular expressions will be discussed in Chapter 4.

MatchCondition : Operand MATCH Operand
                 | Operand NOT MATCH Operand

Figure 2.10: The syntax definition for MATCH conditions

2.6.8 FOR Condition

The FOR condition (Figure 2.11) provides maximum flexibility to test aggregate values. In fact, both the IN and CONTAINS conditions can be expressed in the FOR condition. There are two types of for conditions. One is FOR...ANY condition, the other one is FOR...ANY condition. FOR...ANY condition is satisfied when all the elements satisfy the specified condition. In contrast, the FOR...ANY condition is satisfied if there is at least one element satisfying the condition.

ForCondition : FOR VariableDecl ALL '(' Condition ')' 
              | FOR VariableDecl ANY '(' Condition ')' 

Figure 2.11: The syntax definition for FOR conditions
2.6.9 Compound Condition

All the atomic conditions discussed previously can be combined together to construct new conditions using the logical operators AND, OR, and NOT (Figure 2.3). The NOT operator has the highest precedence, AND is the next, and OR has the lowest. Parentheses can be used to enforce the highest evaluation.

2.7 Select Query

The select query extracts information from a STEP database based on the criteria specified in the where-clause. A select query consists of a select-clause, a from-clause, and a where-clause (optional). The formative syntax is shown in Figure 2.12. As both the from-clause and the where-clause have been discussed, only the select-clause is discussed here.

```
SelectQuery  : SELECT SelectExps FromStat OptionalWhereStat
               | ImpliedSelectStat ImpliedFromStat
               OptionalImpliedWhereStat
               ;
SelectExps   : SelectExp
               | SelectExps ',' SelectExp
               ;
SelectExp    : VariableExtension
               | VariableExtension '(' AttributeLists ')' 
               | VariableExtension '(' '*' ')' 
               ;
VariableExtension : VariablePrefix
               | VariablePrefix '->' AttributeList 
               ;
```

Figure 2.12: The syntax definition for SELECT queries
The select clause determines which part of the information on the collected data needs to be retrieved. It consists of the SELECT keyword followed by a list of expressions separated by commas. When more than one attribute expression needs to be selected and all share the same initial portion in expression paths, the select expression can be concisely written as the common path followed by the list of the rest part of each expression, separated by commas and surrounded by parentheses. For example, “SELECT X->a, X->b.c” can be expressed as “SELECT X(a, b.c)”. If all the attributes of an instance need to be selected, one can use (*) to shorten the expression. Since instance I.D. is a property (not the key attribute of an entity) to identify an object in a database. It must be selected by specifying the select expression as the instance variable only.

Example 1 List all information for all instructors.

EQL> SELECT (*) FROM instructor;

1. 'Bob' 'P' 'Judd' '1347832134' (#10,#20)
2. 'David' 'A' 'Koonce' '4758943275' (#30,#40)
3. 'Charles' 'M' 'Parks' '8432967583' (#50)
4. 'Gray' 'M' 'Graham' '345781235' ()

Example 2 Select all students.

EQL> SELECT (firstname, middlename, lastname) FROM student;

1. 'Julia' 'R' 'Lee'
2. 'Ema' 'R' 'Lee'
3. 'Kevin' 'L' 'Smith'
4. 'Dorothy' 'A' 'King'
5. 'Tom' 'B' 'Tang'
6. 'Larry' $ 'Maher'
7. 'Frank' 'G' 'Whitehead'
Example 3 Select the call numbers and titles for all the classes taught by professor
whose id is '1347832134'.

EQL> SELECT y(callNumber, title)
       FROM x IN instructor, y IN x->classes[]
       WHERE x->id == '1347832134';

1. 'EE690'  'Advanced CIM'
2. 'EE590'  'Programming With C++'

Example 4 Select all classes taken by student whose id is '444444444'.

EQL> SELECT c(callNumber, title)
       FROM s IN student, c IN s->theStudent<<enrollment.theClass
       WHERE s->id == '444444444';

1. 'EE690'  'Advanced CIM'
2. 'EE590'  'Programming With C++'

Example 5 Select all students who have ever taken any classes with the student whose
id is '444444444'.

EQL> SELECT s2(firstname, middlename, lastname)
       FROM s IN student, s2 IN student
       WHERE s2->id != '444444444' AND s->id == '444444444' AND
       FOR c IN
       s->theStudent<<enrollment.theClass.enrollments[].theStudent
       ANY (s2 == c);

1. 'Ema'  'R'  'Lee'
2. 'Larry'  $  'Maher'

2.8 Update Query

The UPDATE query modifies the attribute values of existing instances.
Updating includes: setting an attribute to a new value; inserting/removing elements
into/from an aggregate (if aggregate is order sensitive, then position can be specified);
or setting an element to a new value at a specific position. The structure of the update
query consists of an update-clause, a from-clause, and an optional where-clause. The syntax and semantics for both the from and where clauses are the same with those in select queries. The normative definition for update clause is shown in Figure 2.13.

```
UpdateQuery   : UPDATE_TOKEN VariableName '(' AttributeUpdates ')' 
               FromStat OptionalWhereStat

AttributeUpdates : ADD_TOKEN AttributeAdditions 
                   | SET_TOKEN AttributeAssignments 
                   | DELETE_TOKEN AttributeDeletions 
                   | AttributeUpdates ADD_TOKEN AttributeAdditions 
                   | AttributeUpdates SET_TOKEN AttributeAssignments 
                   | AttributeUpdates DELETE_TOKEN AttributeDeletions

AttributeAdditions : AttributeAddition 
                     | AttributeAdditions ',' AttributeAddition 

AttributeAddition : Value TO_TOKEN SimpleAttribute 
                    | Value AT_TOKEN IndexedAttribute

AttributeDeletions : AttributeDeletion 
                     | AttributeDeletions ',' AttributeDeletion

AttributeDeletion : Value FROM_TOKEN SimpleAttribute 
                    | IndexedAttribute 
                    | ALL_TOKEN Value FROM_TOKEN SimpleAttribute

SimpleAttribute : AttributeName

IndexedAttribute : AttributeName Index
```

Figure 2.13 The syntax definition of an UPDATE query
Example 6 Change the course grade of enrollment #1130 to 99.

EQL> UPDATE x (SET courseGrade = 99)
    FROM x IN (#1130);

Instances updated: 1

Example 7 Delete enrollment #1130 from class "Programming With C++".

EQL> UPDATE X (DELETE #1130 FROM enrollments)
    FROM x IN class_offering
    WHERE X->title == 'Programming With C++';

Instances updated: 1

2.9 Insert Query

The INSERT query inserts a new instance into a STEP model. As Figure 2.14 shows, it consists only of an insert clause. The insertion can be done by the following three methods: (1) insert a new instance without any giving attribute values; (2) insert a new instance given part of the attribute values; (3) specify the values for all the attributes. In the first case, each attribute value is assigned to unknown and in the second case, the unspecified attributes are assigned to unknown. This query returns the I.D. for the new instance.

While the EXPRESS schema requires attributes without the OPTIONAL keyword to assigned values during their lifetimes, Eql has been intentionally designed to tolerate the potential violation of data integrity by methods one and two temporarily at the time of creation. In practice, this feature is more convenient for users to create instances without specifying all information. The trade-off is that users should pay more attention, as Eql will not find the violations of data integrity until the time it commits a model, when it checks using specified integrity measures in the schema.
Figure 2.14 The syntax definition of INSERT query

Example 8 Create a new student without specifying any information.

EQL> insert student();
Instance added: #1980

Example 9 Create a new student whose name is Lizhong Huang and I.D. is 001865137 and has no enrollment.

EQL> insert student ('Lizhong', '', 'Huang', '001865137', 0);
Instance added: #1990
Example 10 Create a new class whose title is "Advanced Software Integration" and other information is unknown.

```
EQL> insert class_offering (title = 'Advanced Software Integration');
Instance added: #269416968
```

2.10 Delete Query

The delete query deletes instances from a STEP data model. It consists of three clauses delete-clause, from-clause, and an optional where-clause. The syntax of a delete query is shown in Figure 2.15.

```
DeleteQuery : DELETE_TOKEN VariableRef
              FromStat OptionalWhereStat
              ;

OptionalWhereStat : WHERE Condition
                   ;

Figure 2.15 The syntax of definition of DELETE queries
```

Example 11 delete a class whose title is "Advanced Software Integration".

```
EQL> DELETE x
    FROM x IN class_offering
    WHERE x->title == 'Advanced Software Integration';

Instances deleted: 1
```
The references to those to-be-deleted instances are not removed automatically at the deletion. Therefore, a user should be careful enough to update all the related instances who have reference to the deleted ones.

2.11 System Commands

In addition to the four different types of data manipulating queries discussed in the previous sections, Eql has several system level commands to manage STEP models on the system level.

Synopsis CREAT `file_name', `schema_file_name';

Description Creates a STEP file named file_name whose schema is stored in schema_file_name. Both arguments should be enclosed into single quotes. file_name can contain the relative or absolute path but should be with no .stp extension. schema_file_name should be without .rose extension, and it is searched in order under the directories specified by searching path environment variable ROSE_DB.

Synopsis OPEN `file_name', ['mode'];

Description Opens the STEP file whose name is file_name with mode. Specification of file_name is the same as in the create command. However, one should be aware that this file is also searched in order under the directories specified by searching path environment variable ROSE_DB. There are two possible modes. One mode is RW (or rw), which sets the opened file to subject both read and write. Another mode is RO (or ro), which sets the opened file for read only. If the STEP model has already been opened, this command simply makes the model be the current one. The default mode is "RW".
**Synopsis** COMMIT ['file_name'];

**Description** Saves all the changes that have been made since last opening or saving for model *file_name*, whichever is more recent. If *file_name* is omitted, it commits the current STEP model. The model which is committed keeps opening status; i.e., the current STEP file remains current after commitment.

**Synopsis** CLOSE 'file_name';

**Description** Closes the STEP model *file_name* which should be already opened without saving. If *file_name* is omitted, it closes the current STEP file. One should not try to close a newly created model; otherwise, behavior is not defined.

**Synopsis** EXIT; or QUIT;

**Description** Exits from the execution of Eql without saving any changes on any opened or created files in the session. Therefore, a user must be careful enough to commit all the files intended to be saved.

**Synopsis** ROLL BACK ['file_name'];

**Description** Undo all changes in model *file_name* since last time reading or saving, whichever happened more recently. If *file_name* is omitted, Roll back undoes all changes in the current model.

**Synopsis** !shell_command;

**Description** Forks a child process to execute the shell_command. This command is occasionally useful when a user does not want to exit from Eql, yet wants to be able to execute a UNIX command. For example, "!date;" prints out the current date. "!ls -F" prints out all the files under the current directory. Notice that, "!cd .." will not change the working directory of its parent, as a child process cannot affect its parent process.
Chapter 3

An Implementation of EXPRESS Query Language

3.1 Introduction

Chapter 2 specified the syntax of Eql. To demonstrate the feasibility, the author presents an EQL interactive program (henceforth, EQL) in this chapter. EQL repetitively reads an Eql statement from standard input, interprets, and executes it. It has been implemented on both IRIX and HP-UNIX. The implementations on other platforms can be achieved similarly if relevant developing resources are available. The implementation has extensively deployed a software package called ST-Developer (v1.5) from STEP Tools, Inc. This package provides several programming resources supporting STEP standards application development, including tools such as EXPRESS compilers, and libraries of various programming languages. The portion applied by EQL is called SDAI C library, which contains a set of C functions for accessing data in the EXPRESS data dictionaries compiled by the compiler in the same package.

EQL of the current version is not able to conform 100% to the specifications in Chapter 2 because of current SDAI C library limitations. So the author lists all the differences between EQL and Eql specifications in a dedicated section after several
sections of technical examinations of the implementation. Last in this chapter, the usage of EQL is introduced and the error codes are explained in detail. As Figure 3.1 illustrates, the interpreting processes of EQL can be divided into three stages. The first stage is lexical analysis using Lex. It takes an Eql query as input and outputs a list of tokens, which are taken as input by the second stage. The second stage uses Yacc to analyze the grammar rules and build up a data structure (a node tree) from the tokens. The final stage is the execution of the node tree. Exceptions detected on the first two stages are compiling errors, while those on the third stage are runtime errors.

![Figure 3.1 EQL has three interpretation stages](image)

**3.2 Overview**

The architecture of EQL is shown in Figure 3.2. EQL accesses data from system data dictionaries, user data dictionaries, and the STEP database via the SDAI C library. The user dictionaries are generated from the compiling of user's EXPRESS schemas. All dictionaries are in ROSE format. The data flows between data dictionaries (of both system and user) and the SDAI C library are in one direction, i.e., SDAI never writes any data into a data dictionary. It accesses them only for extracting information from them (for example, information about an entity definition or an attribute data type). Data communication between SDAI C functions and the STEP
database are in two directions. SDAI not only reads data from a database, but also outputs data into it. Notice that Figure 3.2 is a conceptual architecture. In practice, because of efficiency, SDAI actually loads all the STEP database and part of the data dictionaries into memory in the early stages of a session and accesses this block of memory directly. The chance of using the other portion of data dictionaries is small and will be loaded only if necessary during runtime.

Figure 3.2: The architecture of EQL
3.3 Data Structures

There are two major data structures in the implementation. The first is called V_NODE.

```
struct v_node {
    struct v_node next;
    struct v_node contents;
    struct v_node last;
    Q_TYPE type;
    SdaiADB v;
};
```

This data structure is used to represent an aggregate constant or a selecting result from a select query. The next pointer points to the following element of the aggregate. The contents pointer can be used to point to the next level of an element when representing a nested aggregate value. The last pointer always points to the last node on the same level. It provides more efficiency in appending/removing nodes to/from an aggregate or a query result. Q_TYPE is an enumeration of node types for both V_NODE and Q_NODE (see below). For V_NODE, the value of type can be one of the following: V_ROW, V_COLUMN, V_SCALAR, V_AGGREGATE, and V_UNKNOWN. The contents of a V_ROW node represent the value of a row, which consists of a list of column nodes. Next attribute of a V_ROW node points to the node of the same type. The value associates to a node can be stored in attribute v, which is of a SDAI data type.

The second key data structure is called Q_NODE. This structure corresponds to each node inside the node tree between the second and third stages in Figure 3.1. The context of this node is shown below.
struct q_node {
    struct q_node* delete;
    struct q_node* next;
    struct q_node* contents;
    struct q_node* last;
    struct q_node* ref;
    Q_TYPE type;
    Char* s;
    Long i;
    Double r;
    SdaiADB v;
    SdaiIterator itr;
    struct v_node vn;
    struct v_node vn_itr;
    BOOL status;
};

Attribute `delete` makes a link list of nodes to be deleted (deletion of nodes takes place after execution). When one node receives a "delete" message, before it tries to delete itself, it passes the message to the node `delete` points to. Attributes `next`, `contents` and `last` serve similar purposes as those in `V_NODE`. Pointer `ref` is used by a variable node to point to a proper variable declaration node. Attribute `type` indicates the type of a node. Class and variable nodes store their names in attribute `s`, which is a string pointer. Integer and real constants are stored in attributes `i` and `r`, respectively, while aggregate constants are stored in attribute `vn`. Intermediate values, such as one returned from SDAI library functions, can be held by attribute `v`. Attribute `itr` and `vn_itr` are two iterators to `v` and `vn`, respectively. The last attribute, `status` indicates a dynamic state of some nodes, such as class nodes to route messages. The general node tree structure is shown in Figure 3.3. It starts with a node called "begin". A horizontal pointer in the sketch corresponds to the `next` attribute in the data structure, while a downward pointer corresponds to the `contents` attribute. The nodes with solid lines (e.g. the begin node)
represent the real ones in the implementation; those with dash lines (e.g. the condition node) represent the conceptual nodes, each of which is equivalent to a group of nodes in the implementation. Pointers with dash lines indicate the nodes pointed to are optional; in other words, these pointers may be null pointers.

Figure 3.3: The backbone of the EQL node tree (portion shadowed is not supported by the current version of EQL)
Now let's apply a select query shown in Figure 3.4 as an example to further our discussion. This query prints out the callNumbers and titles of classes which are taught by an instructor whose id equals to '1234'. The node tree constructed for this query is shown in Figure 3.5. In the node tree, there are two nodes with type var_dcl—one is X and the other is Y—corresponding to the two declarations in the query. Notice that there is an element node ([]) below V_NODE instructor because instructor is a class node. Each class node has been implicitly appended by an element node to bust the aggregat of entity instances. The atomic condition node under where is very straightforward. There are two select expression nodes following node where---the first is for callNumber and the second is for title.

```
SELECT Y(callNumber, title)
FROM X in instructor, Y in X->class
WHERE X->id == '1234';
```

Figure 3.4: An example Eql query

The third data structure is related to the usage of host variables. The interactive EQL program does not apply this data structure and the relevant functions. However, the presentation of this data structure makes it possible for third parties to develop Eql conforming software with the host variable feature. The details of the data structure are shown below.
struct hv_node {
    char* name;
    int index;
    long i;
    double d;
    char* s;
    HOST_VAR_TYPE type;
    Q_NODE* qv;
    HV_NODE* next;
};

The name attribute holds the host variable name, and index is used to store the index if the host variable is an array element. Attributes i, d, and s are used to store the host variable values during bindings. Type is an enumerated value to express the host variable type; i.e., integer, real, or the other SDAI built-in data types. Attribute qv points to the proper Q_NODE inside the query node tree.

Figure 3.5: Node tree constructed from the example query in Figure 3.4
3.4 Major Node Constructs

Now that we have analyzed the data structures, in this section, we will discuss a few major node constructs along with their responsibilities. The other node constructs are similar and are too many to be listed completely here. All nodes to be discussed are Q_NODEs. In the coming context, sometimes we refer to them using their type names. For example, a Q_NODE with type q_var_dcl is called var_dcl node or simply var_dcl.

3.4.1 Variable Declaration

Figure 3.6 shows the variable declaration node constructs. Under a var_dcl node, there is either a var_ref or a class node. Under either of these nodes, there is an attr_ext_dcl node, which is optional for a class node. The attr_ext_dcl node is a conceptual and pseudo node, as shown in Figure 3.6. This node consists of a list of one or more of the following nodes: element node ([I]), attr node, and inv_attr_node (linked by Q_NODE contents pointer). Var_dcl node first sends a message to its contents and the contents should bring out the value out of the execution (for example, if var_dcl is declared to be a class, then the value should be the next instance of that class). Also, the function returns an enumerated value: S_V, V_V, or E_L. If the return value is S_V (stands for single value), it means that the execution only generates a scalar value. Var_dcl then sends a message to its next node (the node pointed by next). If returned value is V_V (stand for various values), it first sends a message to its next node, and then sends a message to its contents to ask for the next value and repeats the process until the contents node returns E_L (stands for “end of list”).
Figure 3.6: Nodes for variable declarations
3.4.2 Element Node

The element ([]) node is just one node for its construct (the internal node type is called q_element). An element node can be an explicitly specified one in the Eql query or one implicitly patched by the EQL interpreter. It receives an aggregate as input and busts it into elements to the next level. Although it is just a single node, it has a relatively complex manipulation algorithm. It uses a status to automate the actions. The status can be one of the following: UNINIT, CONTENTS, and SELF with the initial value as UNINIT. With UNINIT status, it initializes itself with the input value and updates its status as SELF. With status SELF, it resolves the next element it has busted and passes the value downward if possible. Otherwise, it returns to its caller immediately. The status can only become CONTENTS from SELF (when it successfully resolved a value and has a content node). With CONTENTS status, it does nothing but passes an execution message to its contents and returns whatever it receives.

3.4.3 Attribute Node

An attribute node corresponds to the attr node in the sketch. Under it, there is zero or more index nodes (Figure 3.6). The input value to an attribute node must be of instance data type; otherwise, runtime error must be raised. It also applies the internal node status (only UNINIT and CONTENTS) to automate the execution. At the beginning of the status is UNINT, and it resolves the corresponding attribute value using the input value and the attribute name which holds by itself. If the resolved attribute value is $, it returns to its caller with S_V value. Otherwise, it changes its status to CONTENTS and passes value down. When its status is CONTENTS, it
simply sends message to its contents node and returns whatever it receives. Its status can be changed from CONTENTS to UNINIT, when its contents return $S_V$ or $E_L$.

### 3.4.4 Inverse Attribute Node

An inverse attribute node consists of an `inv_attr_class` node and an `attr` node. `Attr` node sits on the next `inv_attr_class` and the contents of the `attr` node can be another `attr` node, and so on (Figure 3.6). Similar to element and attribute nodes, this node automates the execution tasks using status state machine. When the status is UNINIT, it collects all the entity instances of its class and sequentially sends each entity instance to its next node as the input value along with an execution message. The attribute node resolves the final attribute value to it. Then the `inv_attr_class` node compares the input value it receives and the returned value from the next node. If these two values are equal, then it saves the instance; otherwise, it removes the instance from the instance list. If the initialization is successful (there is at least one instance left on the list), it changes its status to SELF; otherwise, it returns to its caller directly with proper value. When its status changes to SELF or CONTENTS, it has similar behavior as the element node with the same status value.

### 3.4.5 Condition Nodes

The condition node constructs are illustrated in Figure 3.7. They look a little tedious because of their native natures. A condition node can be an atomic condition or a compound condition (two or more conditions combined by a logical operators NOT, AND, and OR, or a pair of parenthesis. The routine of message is extraordinary simple.
Figure 3.7: Nodes for conditions
The message always goes down first (for the evaluation of the left side of a operator). If the final value returned to where node is true, it sends an execution message to its next node to further action, such as select, update, etc. Otherwise, it does nothing. The condition node can only return either TRUE or FALSE.

3.4.6 For-Condition Node

Among the atomic conditions, the for-condition is worth a separate discussion. The node constructs is shown in Figure 3.8. A q_for_all (or q_for_any) node sends a message to its next node, which is a list of var_dcl nodes. As we have analyzed in Section 3.4.1, var_dcl nodes produce the complete variable combinations, and for each combination sends a message to the node following them—in this case, a q_for_all_where (or a q_for_any_where) node. The q_for_all_where node will test its condition (the condition node on its contents) and if any combination returns false, it will let the var_dcl nodes to stop sending a new combination, while the q_for_any_where node notifies the var_dcl nodes to stop sending new combinations after it is satisfied for the first time.

Figure 3.8: Nodes for for conditions
3.4.7 Select Expression Nodes

A select expression is used in select queries, and its node constructs are sketched in Figure 3.9. The node constructs can be a all_atrib node, pseudo node attrib_ext, or attrib_ext node followed by all_attrib node. As both attr and inv_attr_class nodes have been discussed, we only discuss all_attrib node here. The input to this node is an entity instance or an aggregate whose final elements are entity instances. For each entity instance, all_attrib nodes first retrieve all the attribute names and then, based on the attribute name, resolve the corresponding attribute value. Each attribute value is arranged into the final select result. Notice that this node has neither next or contents node, so it passes no message to anyone. Instead it returns void to its caller.

3.5 Significant Global Variables

The implementation applied more than a dozen global variables and the significant ones are described below.

- **BEGIN_NODE** is a pointer to a Q_NODE, which is the starting node of the constructed query node tree.

- **CommitFile** is the pointer to the output .stp file stream. This global variable is used in a separated module which remedies the commit function bug in SDAI C v1.5.
Figure 3.9: Select expressions
- **Current_model** is a global variable of SDAI built-in data type, SdaiModel. Most of model-related SDAI functions need the handle of the working model, and this variable has been applied by Eql to intelligently manage the multiple models at the same time.

- **Current_v_node** is of V_NODE (pointer to v_node data structure) data type and is used to append coming select results during the execution of a select query.

- **Eql_state_ptr** is a character pointer managed by the Lex code and the eql_parse() function. Eql_parse() function first initialized it to the beginning of the Eql query statement, then Lex increases or deceases it in input() and unput() macros.

- **Error_detected** is an integer flag to switch on/off debugging message during the parsing process. **Host_vars** is a pointer to the list of host variables (in the format of HV_NODE). The list is generated by the parsing module (eql.y) and is sent to a third party to resolve the values of each host variables.

- **InstanceMap** is a 2D array, each row of which contains two long integers. This variable has been designed to map the memory address of an entity to a human readable object I.D.. Again this variable is for the purpose of remedying the buggy commit functions in the SDAI C library.

- **LONG_RESULT** is of long integer data type and represents two distinct meanings for different types of queries. For an insert query, it represents a newly created object I.D. (memory address) and for other types of queries, it represents the number of objects affected.

- **NODES** points to a Q_NODE list which need to be deleted upon certain conditions.
- **Part21_model_list** is an important variable which makes it possible for Eql to intelligently organize the multiple SDAI models at the same time. Each opened or created model has a representative node in the list. And each open or create query needs to consult the list before it performs any actions.

- **QUERY_TYPE** holds an enumerated value indicating a query's type.

- **Query_done** is a flag integer watched by yywrap() function in the Lex code. Initially it is set to false and if an error occurs, its value becomes true.

- **SELECT_RESULTS** is of V_NODE type and is the starting node of the selected results for a query.

### 3.6 Major Routines

The brief descriptions of major routines are given here for reference. Routines related to the entire model are given in the next section.

#### 3.6.1 Manipulation of a Q_NODE

This category of functions are for manipulating Q_NODES, and most of them have very self-explanatory names. Function add_q_node() creates a new Q_NODE and at the time of creation, its contents and next pointers have been set up to the specified values via function parameters. Append_q_next() and append_q_contents() functions append a Q_NODE to one's next or contents, respectively. Node_dup() function duplicates a Q_NODE. Print_q_node() function prints out all the attribute values of a node to the standard output device. This function only bears with debugging purposes.
Function delete_q_nodes() deletes the nodes pointed to by both contents and next pointers first before it deletes itself.

### 3.6.2 Manipulation of a V_NODE

There are several similar functions for a V_NODE, including: add_v_node(), append_v_next(), append_v_contents(), and print_v_nodes(). The function for deleting V_NODE will be discussed in Section 3.7, as it can be possibly used by a third party.

### 3.6.3 Manipulation of Host Variable Nodes

The host variable nodes (HV_NODE) have similar manipulation utilities: add_hv_node(), append_hv_next(), and delete_hv_nodes().

### 3.6.4 Binding Q_NODEs

Remember that some nodes are the reference nodes in the tree (for example, every variable used in an Eql query is a variable reference node except for the one corresponding to its declaration). Therefore, after a Q_NODE tree has been built up and before it is executed, these reference nodes need to be binded. That is, their ref pointers should be linked to where they are declared. This task is fulfilled by function bind_q_node().

### 3.6.5 Execution of Q_NODEs

Corresponding to the third stage in Figure 3.1, there is a big C function exec_q_node() in the implementation. This function is the core part of the EQL
interpreter. It has three parameters: the first one is the pointer to a Q_NODE, which is
the one to be executed. The second pointer is a special SDAI data structure, which is a
value sent to the node, while the last one is a pointer to a data structure of same type,
which is a value to be brought out from the execution (if there is no value coming out,
the pointer will be set to null). The function also returns an enumerated value which
indicates the status of the just-finished execution. We have encountered all of these
values in the previous context; namely, S_V, V_V, E_L, VOID, TRUE, and FALSE. It
uses a big state machine base on the node type to automate the execution. As one may
have expected, this function has been recursively called in a quite high degree---when
one node receives a message, in most of cases, it needs to send another execution
message to other nodes before it executes itself.

### 3.7 Model Interfaces

As stated before, this specific EQL implementation not only provides an interactive
program to perform Eql queries, it also provides third parties a C library for developing
Eql-conforming software. This section lists all the interfaces of the library. All these
function names start with "eql_" except for one.

- **set_eql_debug** sets the switch of debugging. For the historical reasons, the name of
  the function does not follow the naming convention.

- **eql_roll_back_model** rolls back a model to the status of the last saving or
  opening.

- **eql_commit_model** saves the current model permanently.

- **eql_close_model** closes the current model without saving.
• **eql_close_all_models** closes all the models which have been opened without saving. The created models are exceptionally committed.

• **eql_parse** parses the given query to construct the parse node tree and the list of host variables. It returns zero if there is no host variables or parsing error occurs; otherwise, it returns one.

• **eql_print_v_nodes** prints the attributes values of the given V_NODE tree. This function is used for debugging only.

• **eql_create_model** creates a new SDAI user model and sets it as the current mode.

• **eql_open_model** opens or restores a SDAI user model.

• **eql_start** prepares for a SDAI session, including initialization of the necessary variables, opening the session and repository required by SDAI.

• **eql_shut_down** closes all models and cleaning up transactions, repository, etc. by calling relevant SDAI C functions.

• **eql_process** executes the current Q_NODE tree (via global variable BEGIN_NODE) and stores the results into a V_NODE (specified in the parameter). It returns the query type.

• **eql_get_result_array** takes a V_NODE tree as part of the input. It forms an array of string based on the input value and each element string for one row. A string value is the concatenation of all the column value separated by the delimiter (also specified by a function parameter).

• **eql_bind_host_variable** walks through the given host variable node list and sets up the proper Q_NODE to the value the list node currently holds.
3.8 Caveats

The current version of EQL has not been able to fully support all the specifications stated in Chapter 2. This section collects all the caveats of the current EQL implementation. These caveats result from neither the design nor the implementation flaws. Instead, they are caused by the SDAI C library—the implementation tool on which EQL depends. Therefore, these problems will be easily removed once an improved SDAI C library is available in the future.

- EQL does not distinguish set, bag, array, and list data types. The reason is that the current ST Developer tools does not do so. This limitation has triggered the following two limitations.
- EQL does not support the equal operator for aggregate data types. Because EQL can not tell the data type of an aggregate value, it is not able to compare two aggregates sensibly.
- The CONTAINS and IN operator have been limited to work on 1-level aggregate values only. As discussed before, both IN and CONTAINS operators require the comparisons of elements, and because equal operator does not work on aggregate, these two operators has to be limited to work on 1-level aggregate only.
- EQL does not distinguish between NUMERICAL and REAL (both have been treated as real data types). EQL does not support BINARY data type, as ST-Developer seems to have some uncertain patterns with binaries.
- EQL does not support entries of constructs defined in ISO 8859 because ST-Developer does not handle them properly.
• For some data types, EQL cannot set or test the unknown value. These data types are listed in Table 3.1.

• Data type SET OF LOGICAL is read as SET OF BOOLEAN (also true for LIST, BAG, and ARRAY). In other words, value .U. is not an accepted value for the set membership.

• Array index always starts from zero (0), regardless of the schema definition.

• Open command mode “RO” is not effective. For now it equals “RW”; that is, SDAI models opened with “RO” command can still be edited and the changes can be committed permanently.

• Abstract entity type can be instantiated.

• Derived attributes and inverse attributes are not allowed in attribute expressions.

• Non-optional attributes can have unknown values during an object's lifetime.

• Local rules (Uniqueness rule (UNIQUE) and domain rules (WHERE)) are not enforced. The current version of ST-Developer does not support this data integrity checking.

<table>
<thead>
<tr>
<th>Table 3.1: Unknown Values For Some Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Type</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>Integer</td>
</tr>
<tr>
<td>Real</td>
</tr>
<tr>
<td>Logical</td>
</tr>
<tr>
<td>Boolean</td>
</tr>
</tbody>
</table>
3.9 How To Use EQL Interactive Program

3.9.1 Setting Up Environment Variables

EQL itself does not expect any environment variables. However, to make it work properly, one should set up at least one ST-Developer environment variable called ROSE_DB. This environment variable contains all the paths to the STEP databases separated by colon (:). The minimum ROSE_DB value should contain the ST-Developer systems database path. For example, the following command line sets up ROSE_DB to the current directory and the system database of the ST-Developers for the author's setup.

```bash
setenv ROSE_DB ./usr/appl_2/stepools_1.5/system_db/schemas
```

3.9.2 Compilation of EXPRESS schema

Compilation of an EXPRESS schema can be done by the following UNIX command assuming the UNIX standard environment variable PATH has been correctly set up so that file express2sdai is searchable.

```bash
express2sdai -sdai_validate -rose my_schema.exp
```

The command will generate the following three files in ROSE format upon successful compilation.

```bash
my_schema.rose
my_schema_DICTIONARY_DATA.rose
my_schema_PARSE_DATA.rose
```

Of course, the location of these files should be included in the paths specified by ROSE_DB.
3.9.3 Launching Eql Interactive Program

Typing "eql" under UNIX shell prompt will launch the interactive EQL program. After successful starting, the "eql>" prompt will replace shell prompt and the program will be waiting for user input of Eql statements (see Figure 3.10). Each statements can be split into several lines and must be ended by a semicolon (;). Redundant white spaces are all suppressed.

```
%1 eql
  eql>
```

Figure 3.10: EQL prompts "eql>" and waits for user input
Chapter 4

The Templates and Rules Language

4.1 Introduction

Software integration architectures, such as IMDE[26], are difficult to implement using traditional languages, such as C. The most important reason is that these architectures require extensive communication among heterogeneous systems. Usually a script language is the choice in these scenarios[33].

As we have discussed in Chapter 1, a script language has a higher programming level than do system languages. System languages are designed for building data structures and algorithms from scratch with the assumption that the basis of the computing environment is words of memory. Script languages are designed for gluing—they assume the existence of a set of components and are intended primarily for connecting components. Unfortunately, the current existing script languages such as Tcl, Perl, etc., are not very applicable to integration because none of them support all the interfaces to the many and various data resources. In today's software systems, the common data storage methods vary widely and include ASCII text files, relational databases, computer network, as well as other computer terminal devices.
For manufacturing software systems, another important data storage method is a STEP database. In this chapter, a new script language called Tnr is introduced. Compared with other script languages, Tnr not only has strong data analysis ability, but also has the interface capability needed for supporting access and manipulation of different data resources.

4.2 Overview

Tnr is a script language. It includes the typical characteristics of script languages such as loose data typing, garbage collection, etc. The most important feature of Tnr is the direct support of SQL and Eql, so it provides the possibility to develop seamless programs in the heterogeneous computing environment, especially in manufacturing software system integration. Tnr has learned some syntax from other popular programming languages, such as Pascal and C, so that learning Tnr should be a minimal effort.

4.3 Variables

Variables are used to hold temporary values during the execution of a program. Strongly typed programming languages, such as system programming languages, usually support numerous basic built-in and user-defined data types thus, any violation of the rules in the using of variables will be caught on the type checking stage by the compiler. Tnr variables, on the other hand, are loosely-typed. In fact, Tnr only supports two types of variables—scalars and arrays. A scalar can be used to store an integer, a double, or a string. An array can hold a cluster of integers, doubles, strings,
or a mix data of these three types. The index of an array starts from zero. There are
two other issues regarding Tnr variables: one is the declaration and scope; the other is
memory management.

4.3.1 Declaration and Scope

Variables must be declared before being used in Tnr. The declaration of a
variable should start with keyword var and then one or more variable names (separated
by semicolon, if necessary). If a variable name is followed by a pair of brackets([ ]),
then the variable being declared is an array variable. A variable also has its scope: if it
is declared outside of all procedures, it has a global scope; otherwise, it has a local
scope. A global variable is visible and accessible within the whole program, while a
local one is limited into the local procedure. Initialization during the declaration of a
variable is also permitted in Tnr. For example, the following two declarations are valid.

```plaintext
var pi = 3.1415926;
greeting[] = {"Hello", "World", "!"};
```

A variable can be tested to determine whether it is initialized using the function
is_init(). This function expects the name of a variable as its only argument. If the
variable specified by the argument is initialized (associated with a value), it returns
true; otherwise, false.
4.3.2 Memory Storage Management

Memory management is not only tedious but also error-prone. As with many other script languages, Tnr totally eliminates the requirements for a programmer to manage memory (allocation and free-up memories). This feature not only cancels the possibility of memory management errors by users, but also lets a programmer direct more efforts to the implementation of algorithms. For example, a scalar can hold a string of any size. Tnr permits this by internally tracking memory size for a variable and allocating memory when necessary after comparing the string size with the memory size. Likewise, the size of an array appears unlimited to the user. An array variable should not and cannot be specified by its size when it is declared. The memory allocated for variables will be automatically released at the end of a procedure (for local variables) or at the end of the execution of a program (for global variables). There are some occasions when an array size gets very big in the middle of the execution of a program but it is no longer going to be used. It is nice to return this significant resource to the operating system. This can be accomplished by explicitly setting the array size to zero by calling function set_size(). Set_size() has two parameters---the first one is the array variable which needs to be resized and the second is the new size of the array.

4.4 Operators and Expressions

Expressions are the combinations of operands and operators and return new values. The operands can be variables or constants. There are two kinds of constants: numeric constants and string constants. Numeric constants are either integer or real numbers (both floating point numbers and scientific notation numbers). String
constants are sequences of characters surrounded by double quotes (for example "hello world!"). \textit{Tnr} also supports the special character notations defined in C. The complete list of these special characters and their representation are listed in Table 4.1.

Table 4.1: Special Characters

<table>
<thead>
<tr>
<th>Escape character sequence</th>
<th>Replaced By</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>Audible alert (0x7)</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace (0x8)</td>
</tr>
<tr>
<td>\f</td>
<td>Formfeed (0xc)</td>
</tr>
<tr>
<td>\n</td>
<td>New line (0xa)</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return (0xd)</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab (0x9)</td>
</tr>
<tr>
<td>\v</td>
<td>Vertical tab (0xb)</td>
</tr>
<tr>
<td>\</td>
<td>Backslash (0x5c)</td>
</tr>
<tr>
<td>?</td>
<td>Question mark (0x3f)</td>
</tr>
<tr>
<td>'</td>
<td>Single quote (0x27)</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote (0x22)</td>
</tr>
</tbody>
</table>

Tnr operators are divided into two groups: unary and binary operators. An unary operator takes one operand and a binary operator take two operands.

4.4.1 Arithmetic Operators

Arithmetic operators are not limited to numerical expressions. In fact, they can support expressions which result in all three Tnr data types: integer, double, and string. If an operand returns a string, \textit{Tnr} tries to convert it into a numerical value internally. If conversion fails, then \textit{Tnr} reports a runtime error. The conversion rules obeyed by \textit{Tnr} are the following: if a string can be converted into an integer, do the conversion; otherwise, try to convert to a double and if all the characters in a string are consumed by
the conversion, then do the conversion. Otherwise, a runtime error is raised. The conversion does not recognize octal and hexadecimal notations. For example,

10 - "5"       returns integer 5;
10 - "5.1E1"   returns double -41;
10 - "ABC"     causes runtime error;

4.4.2 Comparison Operators

Comparison operators, including <, <=, ==, >=, >, compare their operands as numerical values (string comparison can be done by strcmp() function). If either side cannot be successfully converted into a numerical value, a runtime error is reported. For example,

if (1 < "1.1")  returns true;
if (1 < "1a")   runtime error;

4.4.3 Logical Operators

There are three logical operators: !(not), ||(or) and && (and). They all have the same semantics and behaviors as in C. The evaluation of true and false are also kept the same as in C; i.e., if an expression returns zero, then it is false; otherwise, it is true. For string expressions, conversions are made before evaluations.
4.4.4 String Operator

Tnr supports a special string operator for concatenating two strings. The operator is represented as a vertical bar (|). It converts both operands into two strings (if necessary), concatenates them together, and returns the new string. For example,

"Hello" | "World" | "!"    returns "Hello World!"
"Pi equals to " | 3.14159    returns "Pi equals to 3.14159";

4.4.5 Assigning Operators

Convenient assignment operators in C, such as +=, -=, etc. are all available in Tnr. The side-effects of all these numerical assignment operators is that they all potentially change their operands into numeric types. In addition, Tnr supports |=. The semantics of this operator is similar with others in this group. It concatenates both operands together as strings and assigns it to the left one.

4.4.6 Other Operators

The conventional operators ++ and -- found in both C and C++ are also available in Tnr. In addition, there are a few special operators supported by Tnr which do not have equivalents in C. # is an unary operator taking an array variable as its operand. It returns the array size. ${\{\}}$ operator takes a string expression as its only operand (located inside {}). It first resolves the expression value and then calls for a variable or procedure whose name matches the resolved string value. For example,
template main()
var var_name = "myarray";
myarray[];
{
    set_size(${var_name}[], 10);
    error("size of myarray[] is ", #myarray[], ".n");
}

produces output as

size of myarray[] is 10

Table 4.2 summarizes all operators supported by Tnr. The precedence are in decreasing order. The operators in the same group (between lines) have the same precedence.

4.5 Control Flow

Tnr supports all the execution control flow facilities in C. These control flow statements include the if, for loop, while loop, and switch statements. Their semantics are also kept.

4.5.1 If Statement

The if statement conditionally executes a block of statements. Consider the following example:

if (x == 0) {
    error("x equals to zero, bail out\n");
    exit;
}
Table 4.2: Operators Procedences

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Result</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = B</td>
<td>Assign the value of B to A</td>
<td>Assignment operator</td>
</tr>
<tr>
<td>A += B</td>
<td>Same as A = A + B</td>
<td>Same</td>
</tr>
<tr>
<td>A -= B</td>
<td>Same as A = A - B</td>
<td>Same</td>
</tr>
<tr>
<td>A *= B</td>
<td>Same as A = A * B</td>
<td>Same</td>
</tr>
<tr>
<td>A /= B</td>
<td>Same as A = A / B</td>
<td>Same</td>
</tr>
<tr>
<td>A %= B</td>
<td>Same as A = A % B</td>
<td>Same</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Same as A = A</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Return the concatenated string of A and B</td>
</tr>
<tr>
<td>E1</td>
<td></td>
<td>E2</td>
</tr>
<tr>
<td>E1 &amp;&amp; E2</td>
<td>If both true, return true, otherwise, false</td>
<td>Same</td>
</tr>
<tr>
<td>A == B</td>
<td>1 If A equals to B, otherwise, 0</td>
<td>Comparison operator</td>
</tr>
<tr>
<td>A != B</td>
<td>1 If A does not equal to B, otherwise, 0</td>
<td>Same</td>
</tr>
<tr>
<td>A &lt; B</td>
<td>1 If A is less than B, otherwise 0</td>
<td>Same</td>
</tr>
<tr>
<td>A &lt;= B</td>
<td>1 If A is less than or equal to B, otherwise 0</td>
<td>Same</td>
</tr>
<tr>
<td>A &gt; B</td>
<td>1 If A is greater than or equal to B, otherwise 0</td>
<td>Same</td>
</tr>
<tr>
<td>A &gt;= B</td>
<td>1 If A is greater than or equal to B, otherwise 0</td>
<td>Same</td>
</tr>
<tr>
<td>A + B</td>
<td>Summation of A and B</td>
<td>Arithmetic operator</td>
</tr>
<tr>
<td>A - B</td>
<td>Subtraction of B from A</td>
<td>Same</td>
</tr>
<tr>
<td>A * B</td>
<td>Product of A and B</td>
<td>Same</td>
</tr>
<tr>
<td>A / B</td>
<td>Dividend of A and B</td>
<td>Same</td>
</tr>
<tr>
<td>A % B</td>
<td>Modulus of A and B</td>
<td>Same</td>
</tr>
<tr>
<td>!A</td>
<td>Not A</td>
<td>Logical operator</td>
</tr>
<tr>
<td>&amp;A</td>
<td>Call by reference</td>
<td>Reference operator</td>
</tr>
<tr>
<td>$A[]</td>
<td>Return the array size of A</td>
<td>Array operator</td>
</tr>
<tr>
<td>++A</td>
<td>Increase A by one and return new value</td>
<td>Arithmetic operator</td>
</tr>
<tr>
<td>--A</td>
<td>Decrease A by one and return new value</td>
<td>Same</td>
</tr>
<tr>
<td>A++</td>
<td>Return A and then increase A by one</td>
<td>Same</td>
</tr>
<tr>
<td>A--</td>
<td>Return A and then decrease A by one</td>
<td>Same</td>
</tr>
</tbody>
</table>

The expression x==0 returns either true or false based on x's value. If it returns true, then the two lines inside {} are executed. If statement can be followed by an optional else statement. If the test condition fails, then the statements belong to else are executed. The following example continuously tests the value of x until one is satisfied.
if (x < 0) {
    error("x is less than zero\n");
} else if (x == 0) {
    error("x is equal to zero\n");
} else {
    error("x is greater than zero\n");
}

4.5.2 Loops

Loops repetitively execute a block of statements until some specified conditions are met. There are two types of loops: while-loops and for-loops. A while-loop executes its action block as long as the condition is satisfied. For example, the following example calculates the sum of all the integers from 1 to 100 and stores it in variable sum.

    sum = 0;
    i = 1;
    while (i <= 100) {
        sum += i;
        i++;
    }

Another form of while-loop is shown in the next example. This program executes the action block until the condition is no longer satisfied. The following example keeps asking a user to input a string that contains at least a character.

    do {
        error("Please input your file name: ");
        getline(s);
    } while (length(s) == 0);

The for-loop does not create more functionality than while-loop, but in some cases it is more convenient to use. For example, when one knows the number of times
to repeat an execution, for-loop can be used. The example for calculation of summation
from 1 to 100 can be done by the following for-loop:

```c
for(sum = 0, i = 1; i <= 100; i++) {
    sum += i;
}
```

4.5.3 Switch Statement

The switch statement sequentially compares a specified expression's value with
a list of expressions and executes its following statements if the comparison returns true.

```c
switch (x) {
    case "Turning":
        ...
        break;
    case "Milling":
        ...
        break;
    default:
        ...
        break;
}
```

The comparison between the switch-expression and a case-expression obeys the
following rules. If the switch-expression is a string type, then the comparison is based
on string data type. If the case-expression is not of string data type or the two strings do
not match, it returns false; otherwise, it returns true. If the switch-expression is of
numerical data type, then the comparison returns true, when the case-expression is of
numerical data type and two values are equal.
### 4.5.4 Break and Continue

The break statement can be used inside the block of any loop. If this statement is reached, the next statement to be executed by Tnr is the first one after the block. The continue statement is used in the same manner. When it is used in a while-loop, it makes the next execution the beginning of the block. When it is used in a switch statement, it directly executes the next case block without considering the comparison.

### 4.6 Procedures

As stated before, Tnr is a functional programming language. A Tnr program always consists of procedures with a minimum program containing only one procedure called main. Information processing usually contains two major activities: rules matching and actions. Traditional information processing languages, such as Awk directly support the separation of these two activities. Tnr follows this tradition by providing two types of procedures: rules and templates. The definition of a procedure starts with the keyword rules and templates followed by a procedure name and a pair of parentheses. An optional parameter list to the procedure can be placed inside the parentheses. A procedure can return an optional scalar value. If an array is what is intended as output from a procedure, one must use the call by reference mechanism, which is discussed in the following section.

#### 4.6.1 Calling a Procedure

It has been stated that Tnr is a loosely typed language in terms of variable usage. Also Tnr does not require the number of arguments for calling a procedure to
exactly match the number of parameters in a procedure definition. As long as the argument number is not greater than the parameter number and uninitialized parameters are not called, Tnr does not report the error. A default value can also be specified for parameters in place of a definition, which guarantees the initialization of a parameter. An argument can be passed by reference or by value: if & is placed before an argument, it is passed by reference; otherwise, it is passed by value.

```plaintext
rule init(p1, p2[], p3) //initialize variables
{
    p1 = 3;
p2[0] = "Hello ";
p3 = 1.5;
}

rule dump(p1, p2[], p3 = 0) // dump out variable values
{
    error("p1 = ", p1, "n");
    error("p2[] = ", p2[], "n");
    error("p3 = ", p3, "n");
}

rule main()

    var arg1; arg2[]; arg3;
{
    init(&arg1, &arg2[], &arg3); // call by reference
dump(arg1, arg2[], arg3);   // call by value
    error("n");
dump(arg1, arg2[]); // default value is effective
    error("n");
dump(); /* bad call because parameter p1 of dump()
does not have default value */
```
produces the following output.

```
p1 = 3
p2[] = Hello World!
p3 = 1.5

p1 = 3
p2[] = Hello World!
p3 = 0

p1 =

TNR error -- test.tnr: line: 10: Variable not previously assigned (1)
```

### 4.6.2 Command Line Arguments

The arguments in a command line launching a Tnr program can be passed into the program with the aid of an optional list of parameters in the main procedure. Command line arguments are specified by the "-v" option. Each option can have one or more arguments (separated by commas, if necessary) and each command line can have as many options as needed. Command line arguments are assigned to the parameters in order. The assignment stops upon the consumption of either parameters or arguments, whichever happens first. Tnr complains only under one condition---calling an uninitialized parameter variable. For the following procedure

```plaintext
rule main(arg1, arg2)
{
    error("arg1 = ", arg1, "\n");
    error("arg2 = ", arg2, "\n");
}
```
Command line “tnr -f test.tnr -v 1 -v Hi” or “tnr -f test.tnr -v1,Hi” will generate the following output:

```
arg1 = 1
arg2 = Hi
```

If an array parameter is used, all the parameters behind it will be ignored. This may look surprising at first glance, but it makes sense when we notice that the size of an array variable in Tnr is virtually infinite.

4.7 Data Analysis

Data analysis in Tnr can be categorized into two groups—one mathematical analysis and text (string) analysis. Most of these functions can be found in Awk. As a matter of fact, Tnr can be thought of a superset of Awk with a few exceptions.

4.7.1 Mathematical functions

Mathematical functions supported by Tnr are listed below:

**Synopsis atan2 (y, x)**

**Description** Returns the arctangent of y/x in the range -pi to pi using the signs of both arguments to determine the quadrant of the return value.

**Synopsis cos (x)**

**Description** Returns the cos of the radian argument x.

**Synopsis double (x)**

**Description** Returns the double value of x.
Synopsis \( \exp (x) \)

**Description** Returns the exponential of \( x \).

Synopsis \( \text{int} (x) \)

**Description** Returns the corresponding integer portion of expression \( x \). For example, \( \text{int}(10.52) \) returns 10.

Synopsis \( \log (x) \)

**Description** Returns the natural logarithm of \( x \).

Synopsis \( \text{pow}(x, y) \)

**Description** Returns the power of \( x \) to \( y \).

Synopsis \( \sin (x) \)

**Description** Returns the sin of the radian argument \( x \).

Synopsis \( \sqrt{x} \)

**Description** Returns the square root of \( x \).

Synopsis \( \tan (x) \)

**Description** Returns the tan of the radian argument \( x \).

4.7.2 String Manipulation

Text processing is another important area of data analysis. Tnr provides many functions supporting string analysis and manipulation. These functions are closely related to regular expressions. Regular expressions have long been recognized as a simple yet powerful tool to specify and search character patterns. The regular expressions supported by Tnr are the same as those supported by the match operator in
Eql query. The usages are listed in Table 4.3. The first category includes the search and substitution functions.

Table 4.3: Regular Expressions For Match Condition (this table is adapted from UNIX manual page on IRIX 5.3)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\c</td>
<td>the character \c where \c is not a special character.</td>
</tr>
<tr>
<td>^</td>
<td>the beginning of the string being compared.</td>
</tr>
<tr>
<td>$</td>
<td>the end of the string being compared.</td>
</tr>
<tr>
<td>.</td>
<td>any character in the input.</td>
</tr>
<tr>
<td>[s]</td>
<td>any character in the set s, where s is a sequence of characters and/or a range of characters, for example, [c-c].</td>
</tr>
<tr>
<td>[^s]</td>
<td>any character not in the set s, where s is defined as above.</td>
</tr>
<tr>
<td>r*</td>
<td>zero or more successive occurrences of the regular expression r.</td>
</tr>
<tr>
<td>rx</td>
<td>the occurrence of regular expression r followed by the occurrence</td>
</tr>
<tr>
<td>r{m,n}</td>
<td>any number of m through n successive occurrences of the regular expression r. The regular expression r{m} matches exactly m occurrences; r{m,}, matches at least m occurrences.</td>
</tr>
<tr>
<td>(r^n)</td>
<td>the regular expression r. When (n (where n is a number greater than zero) appears in a constructed regular expression, it stands for the regular expression x where x is the nth regular expression enclosed in ( and ) that appeared earlier in the constructed regular expression. For example, (x(y))2 is the concatenation of regular expressions rxyzy.</td>
</tr>
</tbody>
</table>

**Synopsis** `index(s, t)`

**Description** Returns the position in string s where string t first occurs or 0 if it does not occur at all.

**Synopsis** `match (s, rexp)`

**Description** Returns the position in string s where the regular expression rexp occurs or 0 if it does not occur at all.
Synopsis extract \((s, n, rexp)\)

**Description** Consecutively tries regular expression \(rexp\) in \(s\) for \(n\) times. Upon success, returns the last matched string in \(s\). Otherwise, returns an empty string.

Synopsis **sub** \((rexp, ns, s)\)

**Description** Substitutes the first sub string of string \(s\), which matches regular expression \(rexp\), by new string \(ns\) and returns 0 if failed and 1 if succeeded.

Synopsis **gsub** \((rexp, ns, s)\)

**Description** Behaves like sub() except that it replaces all sub strings which match \(rexp\). It returns the number of successful substitutions.

The second category contains two groups of complementary functions: merge—assembles tokens (strings) into a single string using delimiter and split—splits a single string into tokens based on delimiter.

Synopsis **delim** \((exp)\)

**Description** Overwrites the default delimiter value to \(exp\) and returns the former value. The default delimiter value is used by merge(), split(), eql_exec(), sql_exec(). If this function is never called, the default delimiter value is character FS (value 28) in the ASCII table.

Synopsis **merge** \((s, v1 [, v2, ...vn])\)

**Description** Converts all the args \(v1, v2, ...\) to strings and then concatenates them using the default delimiter (see delim()) to forma new string and assigns it to \(s\). It returns the length of the result string. If an argument is an array, it converts and concatenates each of its elements sequentially.
Synopsis merged (s, delim, vl [, v2, ...])

Description Similar to merge() except that this function uses the explicitly specified delimiter delim as field separator.

Synopsis split (s, vl [, v2, ...])

Description Splits the string s using the default delimiter (see delim() function). The string tokens generated are stored into arguments vl, v2, ... sequentially. If an argument is an array, it will be filled with all the unfilled split items starting from the first element. A runtime error is reported if there are not enough arguments; the number of the split items is returned upon success.

Synopsis splitd (s, delim, vl [, v2, ...])

Description Similar to split() except for it uses delim as the delimiter.

4.7.3 Others

A few more string functions which are helpful.

Synopsis string (s)

Description Returns the string format of expression s. For example, if s holds the numerical value in scientific notation, 5.32, the corresponding string format will be “5.32”.

Synopsis strcmp (s1, s2)

Description Returns a value less than, equal to, or greater than zero indicating the string comparing result of s1 and s2.
Synopsis substr (s, m [, n])

Description Returns the n-byte sub-string of s that begins at position m. If n is omitted, returns the right sub-string of s from position m. The first character of string is at position 1.

Synopsis tolower (s)

Description Converts all upper-case alphabetic characters in string s to lower-case and returns the result string. Numbers and other characters are not affected.

Synopsis toupper (s)

Description Converts all lower-case alphabetic characters in string s to upper-case and returns the result string. Numbers and other characters are not affected.

Synopsis sscanf (s, f [, vl, ...vn])

Description Reads from the character string s based on format string f. Format string f is the same with scanf(). It returns the number of items which have been successfully scanned. (For more information see scanf().)

4.8 Accessing Data Resources

Tnr has been invented mainly for dealing with many and various data resources in heterogeneous and distribute environment. Tnr provides utilities to support data access and manipulation on the following data resources: text files, relational databases, STEP files, networks, and transient data from execution of commands.
4.8.1 Files

Normal text files are a simple and common data storage method. System programming languages support random access to text files. However, most of the script languages only allow forward reading because they only intend to “scan” the text and to match rules. In this section, we discuss all the functions supporting text files accessing in Tnr. The first function is file(). This function opens a file for reading or writing, depending on where it is used. When it is used in rule procedures, it opens a file for reading if the file is not opened yet or restores the file as current input file if the file is already open. When the function is used in a template, it has the same behavior except that it opens or restores a file for writing. A runtime error is reported when file() cannot open a file.

**Synopsis** getline(s)

**Description** Reads from the current position in current file until the new line character or end of file and stores the value into s. It returns the number of characters successfully read or -1, if it reads EOF before reading any other character.

**Synopsis** scan(v1 [, vl, ..., vn])

**Description** Can only be called in rule procedures. It reads from the current input file as scanf() using “%s%s...%s” (see below). It returns the number of items which have been successfully scanned. The arguments can be arrays, in which case all arguments after the array are suppressed because the array size is automatically increased for all remaining tokens. This is also true for scanf(). If no file has been opened for input, it reads from the standard input device.
Synopsis `scanf(f, vl [, vl, ..., vn])`

**Description** Can only be called in rule procedures. It reads from the current input file according to format `f` and returns the number of items which have been successfully scanned. The format `f` is like that for the C function `scanf` except it is simplified. It does not support conversion modifiers `L` and `LL`. Also, it does not support conversions with `$` which specifies the positions of the arguments. If no file has been opened for input, it reads from the standard input.

### 4.8.2 Regular Expression Related Functions

**Synopsis** `move(rexp [, n])`

**Description** Moves the file pointer of the current input file to the end of the `n`th place which matches regular expression `rexp`. Default value for `n` is 1. If matching fails, no movement will be made. Return 0 upon success; otherwise -1. Note that the function cannot apply to stream file (e.g., standard input and pipe).

**Synopsis** `select_rexp(r1[, r2, ... rn] [, first | longest | move_first | move_longest])`

**Description** This function reads from the current input file (undefined for stream files, such as standard input or pipe) line by line, and tries to match regular expressions on each of these lines. The regular expressions are given by the list `r1`, `r2`, ... Notice that matching targets cannot cross lines as the input unit is a line. However, the behavior of input can be changed by `select_rexp_delim()`. If the specified matching rule is satisfied, the matched string can be retrieved by function `last_rexp()`, and the file pointer moves to the next character after the end of the matching string. The function returns the index to the matched regular expression upon success. Otherwise, it returns zero and restores
the input file pointer. The matching rule is specified by the last arguments, which are listed below. The default match rule is first.

- **first** consecutively tries regular expressions until one succeeds. The matching string must start from the current file pointer.

- **longest** tries all regular expressions and takes the longest matching string. The matching string must start from the current file pointer.

- **move_first** tries all regular expressions and takes the matching string which is at the least file position and returns the index to that regular expression. If more than one regular expression have the same least file positions, the first one in the list is selected. The matching string can start from anywhere between the current file pointer and the end of the file. Be aware that the match string cannot cross lines defined by the select_rexp_delim().

- **move_longest** tries all regular expressions and takes the matching string which is at the least file position, and returns the index to that regular expression. If more than one regular expression have the same least file positions, the one with the longest matching length is selected; if there are still more than one expression, the first one in the list is selected. The matching string can start from anywhere between the current file pointer and the end of the file.

**Synopsis** select_rexp_delim([exp])

**Description** Sets up the input unit for select_rexp(). In another words, select_rexp() tries to read the characters between delimiters specified by exp into the buffer each time and tries regular expressions. The delimiting is a normal character string enclosed in brackets, which is applied by C function scanf() as a convention to match a nonempty
sequence of characters from a set of expected characters. (For more information, please reference any ANSI C text.) If this function is never called, the input unit of select_rexp() is a natural line. If exp is not present or is an empty string, the next input line for select_rexp() will be all unread characters in the input file.

Synopsis last_exp()

Description This function returns a string which represent the most recent matched string occurred in the execution of function select_rexp(). If such a string does not exist, it returns an empty string.

4.8.3 Databases

There are two functions supporting the communication to a relational database server. The first function is called db(). This function creates the connection between a Tnr program and a specified database server. The current version of Tnr only supports the communication with Oracle SQL server. The second function is called sql_exec(). This function is used to send an SQL statement to the current connected SQL server and makes it execute the command.

Synopsis db (db_type, db_info)

Description Tnr permits multiple database connections. This function creates a new connection to an SQL server if the specified one does not exist or sets the existing connection to current db_type is a string which specifies the server type. Currently, ORACLE SQL server is the only type supported by Tnr (the corresponding db_type value is "ORACLE"). Each SQL server has its own format specification to request a connection, and this can be specified by argument db_info. The format for ORACLE
SQL server is "user/passwd@T:host:database" (T can be D if DEC net protocol is used to make a connection).

**Synopsis** sql_exec (s[, a[ ] [, delim]])

**Description** Executes the basic SQL statement given by s on the current data base connection. If the optional argument a[ ] is given, the potential query results are stored inside it---one element for one row. The rows are formed by concatenating the column values separated by a delimiter, which is specified by the third argument. If this argument is not present, the concatenation applied the delimiter set by function delim(). This function returns the number of rows the query affected.

**Synopsis** sql_trim (exp)

**Description** Generally, an SQL server patches white spaces after each column value in the query result to produce alignment. However, this feature is not always a good idea, especially in an embedded system. Tnr provides this function to turn on/off this feature of SQL. If the exp is true, the trimmer is one; otherwise, it is off. If the function is never called, the trimmer is on.

### 4.8.4 Host Variables

The SQL statement sent to an SQL server can contain host variables as SQL embedded in other host languages, like C and Pascal. However, the usage of host variables in Tnr is in a one-way direction only; namely, these variables can only be used to bring a value into an SQL server, but not to bring values out. An array cannot be used as a host variable, while individual elements are allowed. Tnr is responsible for any necessary data type translation.
4.8.5 STEP files

The interfaces to STEP files are similar to those for a relational database. There are two major functions to support the activities: p21() and eql_exec().

Synopsis p21 (file_name [, schema_name])

Description If schema_name is present, p21() creates a part 21 model whose name is file_name using the EXPRESS schema schema_name; otherwise, it opens a part 21 model called file_name if the model is not open or sets the model to be current if already open. When it tries to create a new model and a model with the same name exits, a runtime error is generated. Notice that this is the only legal way to create or open a model. The corresponding Eql statements are disabled. However, an Eql server accepts ROLL BACK, COMMIT, and EXIT statements.

Synopsis eql_exec (s [, a/] [, delim])

Description Execute an Eql query given by s. If a/ is present, the result of the query is placed into it---one array element containing one row. Each row is a concatenated string using delimiters. The delimiter is set by the third argument or, if it is does not present, it is the one set by function delim. This functions returns the number of rows for a select query, the number of instances effected for a delete or update query, or the instance ID for an insert query. Host variables can also be embedded into Eql queries. The usage is similar to those in SQL query.

Synopsis eql_trim (exp)

Description Similar with sql_trim(), turn on/off the trimer for eql_exec().
4.8.6 Networks

**Synopsis** socket (*host*, *port*)

**Description** Is the function to be used to access data via internet. It creates a socket at *port* on *host*. When it is called in rule, it creates a socket for input; when it is called in a template, it creates a socket for output.

4.8.7 Transient Data Access

It is also feasible to get transient data from the execution of a command. This ability is accomplished by pipe() function. It creates pipes between the calling Tnr program and the command to execute and sets both input and output files to the streams associated with the pipe. If the same command is called twice by this function, the second time will only set the I/O files in the same way as the first time. Tnr will not execute the command twice.

4.9 Miscellaneous

4.9.1 Literal

Literal, defined as "to print as it is", can appear anywhere inside a program but has to start with a single quote('). If a backslash character is placed at the end of a literal line, the carriage return is suppressed. Literals are useful for printing trivial and format-sensitive string constants; e.g., some constant part in the header section of a data file.
4.9.2 Comments

Tnr supports the syntax for comments in C++. That is, anything in a pair of /* and */ are comments. This pair can appear anywhere except inside a string (between double quotes). In addition, a comment line can be placed after //. The following are two examples which show the two different syntax of comments:

/* set x to 1.0 */
x = 1.0; // this is a comment too

4.9.3 Environment Variable

Tnr provides a function called env(), which can be used to get the value of an environment variable. It expects one argument corresponding to the name of an environment variable and returns the string value of that variable. For example,

error(env("HOME"));

prints out the whole path of a user's home directory. Like

/usr/people/research/huang

4.9.4 Process

Process control not only makes it possible to launch a command, but Tnr provides the means to get the transitional data from it. There are four functions related to this subject. The first function is fork(). This function forks a child process to execute the file specified by cmd and returns the child process ID (PID). The file will be searched from the user's environment variable PATH.
Second function is `kill()`. It kills the process whose PID is specified in the argument. Upon success it returns 0; otherwise, it returns -1.

`System()` is another function to launch a new process. But the parent process will be waiting for it to finish. It also returns standard output from execution. For example, the code

```plaintext
s = system("date");
errorf("The current time is %s\n", s);
```

will give a standard error message as the follows:

```
The current time is Fri May 8 12:10:22 EDT 1998
```

The last function for process controlling is `pipe()`, which has been discussed in Section 4.8.

### 4.9.5 Modularization of Program

When a computer program is getting too big to manage in a single file, it is convenient to separate it into numerous modules, each of which contains a set of relatively close functionalities. When a program is executed, all the files should be listed in the command line and order does not matter.

```plaintext
tnr -f file1.tnr file2.tnr file3.tnr
```

Tnr also supports the include keyword to force the Tnr compiler to read a file specified by its argument when it reaches this statement. This statement can be used anywhere inside a Tnr program (it can, but does not make much sense to use it inside a
procedure). Obviously, this provides a programmer another method to module his/her programs.
Chapter 5

The Implementation of Templates and Rules Language

5.1 Overview

This chapter presents a Tnr interpreter implemented on both IRIX and HP-UX. Today most interpreters and compilers are implemented by compiler-construction tools, such as Lex and Yacc, as "...defining a language, and building a compiler for it using a compiler/compiler seems like the only sensible way to do business."[4] The Tnr interpretation processes are sketched in Figure 5.1. Because Tnr is a loosely typed language, the compiler of Tnr does little about static checking. The regular static checking, such as type checking, flow-of-control checking, and named-related checking, are ignored by the Tnr interpreter. Tnr performs uniqueness checking only on the compiling stage.

Figure 5.1 Tnr has three interpretation stages
5.2 Internal Data Structures

The Tnr interpreter views a Tnr program in terms of symbol nodes and parsing nodes. A SYMBOL_NODE represents a procedure (a template or a rule), a variable declaration, or a parameter; and a PARSE_NODE represents all other language constructs.

5.2.1 Symbol Node

A SYMBOL_NODE has quite a few fields. Its data structure is shown below.

```c
struct symbol_node {
    char* name;
    int type;
    SYMBOL_NODE* var_table;
    SYMBOL_NODE* param_table;
    PARSE_NODE* parse_tree;
    int var_type;
    int defsize;
    int numdefs;
    SYMBOL_VAL** def;
    VARIABLE* val;
    VARIABLE* new_val;
    SYMBOL_NODE* next;
};
```

The first field name stores the name of a procedure, a variable, or a parameter. Filed type has one of the following enumerated values for indicating its node type: TEMPLATE_NODE, RULE_NODE, GLOBAL_VAR_NODE, VAR_NODE, and PARAM_NODE. The next two fields, var_table and param_table, are two pointers to SYMBOL_NODE, respectively. Var_table points to the list of local variables of a procedure. Param_table points to the parameter list of a procedure. Field parse_tree—a pointer to PARSE_NODE—points to all the execution statements of a
procedure in the format of PARSE_NODE. PARSE_NODE will be discussed in the next subsection. The general node tree is sketched in Figure 5.2. Remembering that a SYMBOL_NODE also represents a variable or a parameter, the other fields are needed to keep track of the information regarding variables (perhaps in the future this data structure should be split into two distinct structures to implement each of these two functionalities). The first one is var_type, which can be one of the two enumerated values: SCALAR and ARRAY. Defsize is an integer value of the capacity allocated for an array (it can grow as necessary). Num_defs is an integer keeping track of the number of elements for a default value (e.g., the initialization value of a local variable). Def is an array of pointers to the initialized values, which are represented using data structure SYMBOL_VAL. As shown below, SYMBOL_VAL can store any type of values (integer, double, or string). It is possible to union attributes d, i, and s. However, we decided not to do so to avoid the memory reallocation during internal data conversion.

```c
struct symbol_val {
    int type;
    double d;
    long i;
    int str_size;
    char* s;
};
```

The last two attributes—val and new_val—of SYMBOL_NODE are devised for procedure calls (include recursive calls), which will be discussed in more detail in Section 5.5.7.
Figure 5.2: The node tree of a Tnr program
5.2.2 Parse Node

PARSE_NODE represents each parsing token other than procedures, variable declarations, and parameters after the first stage, as shown in Figure 5.1.

```c
struct parse_node {
    struct parse_node* next;
    struct parse_node* contents;
    struct parse_node* last;
    int type;
    int line_num;
    int file;
    SYMBOL_VAL val;
    struct symbol_node* var;
};
```

Attributes `next` and `contents` are used to link the nodes into a node tree. During the execution, messages can pass along both directions. `Last` attribute always points to the last node of the node list indicated by `next`. It makes the appending nodes more efficient. Attributes `type` indicates the node's type. In total there are more than 100 enumerated node types. Integer attributes `line_num` and `file` store information regarding the location of the PARSE_NODE for reporting run-time errors. `Line_num` is the code line, and `file` is an index, which can be used by error reporting function to resolve the file name. A PARSE_NODE may very well contain a value (e.g., a constant). This value is stored in attribute `val`, which is of SYMBOL_VAL data type. The variables which appear inside the execution statements should always refer to the SYMBOL_NODE in the variable table or parameter table or the global variable list. This reference is built by attribute `var`, which is a pointer to SYMBOL_NODE.
5.2.3 Variable

In addition to SYMBOL_NODE and PARSE_NODE, the implementation was aided by another important data structure—VARIABLE. Its structure is shown in the next page.

```
struct variable {
    struct variable*   next;
    int               size;
    int               num_vals;
    int               var_index;
    SYMBOL_VAL**      val;
    struct variable*  var;
};
```

VARIABLE is a conceptual notation for a variable or a parameter. This concept has been used to represent the value of a variable and is associated with a SYMBOL_NODE. It will be seen in Section 5.5.7 that this data structure is used extensively in procedure calls.

Figure 5.3 shows the pictorial structure of a VARIABLE. Attribute size and num_vals stores the capacity and the actual size of an array variable (an array can regrow upon necessary), while val is the array of pointers to SYMBOL_VALs. As we will see later, when an argument is called by reference, the implementation requires that a VARIABLE be linked to another VARIABLE or an element of another array VARIABLE. This linkage is specified by field var, which is a pointer to VARIABLE, and field var_index, which is an integer index to the element. The last attribute of VARIABLE data structure is called next, which is a pointer to a peer. This attribute is used to build up a VARIABLE list. To accomplish recursive calls, a mechanism of
variable stack for each parameter and local variable is needed. This issue will be discussed in more detail in Section 5.5.7.

Figure 5.3: Variable data structure

5.3 Major Parse Node Constructs

Figure 5.2 has shown the overall node tree constructs. In this section, we concentrate on the parsing node tree constructs. A Tnr program consists of procedures, a procedure consists of statements, a statement consists of expressions, and so on. All the different levels of language elements can be expressed completely by the parse nodes.
5.3.1 Statement Nodes

Each statement has a statement node and these nodes are constructed in the order of the statements: the parse node for the next statement appears on the next side of the current node in the node list. Some statements may have contents nodes but others may not. For example, a literal, a break, and a matched statement do not have any contents nodes. However, the if statement does have contents. We will detail the statement constructs in the next section.

When a statement node receives an execution message, it usually sends message down to its contents if it has one, then performs some functions, and finally passes execution to its next statement node. Each statement nodes returns a parse value—it either decides itself what needs to return or returns whatever its next nodes returns.

5.3.2 Types of Statement Nodes

- **Literal** statements have no contents nodes. The literal value to be printed (PARSE_VAL) is stored inside the node.
- **Expression statements** have contents nodes which correspond to expression. We will discuss expression nodes in details in the next section.
- **Terminal statements** include break, continue, return, matched, failed, and exit statements. Return and exit statements have optional contents nodes for the returning or exiting values, respectively.
- **Null statement** has only a null statement node. It does nothing except for passing a message or returning a value.
• **Do statement** has a syntax like do cmd_stat while (exp) format. The node constructs are shown in Figure 5.4.a. The do statement has similar behavior as the while statement.

• **While statement** has a syntax like while (exp) cmd_stat format. The node constructs are shown in Figure 5.4.b. The while statement node keeps executing its contents nodes until one of the following condition is met: the terminal statement in the command statements is executed or the exp returns false. For the first condition, if the statement is return or exit, the while statements returns whatever the terminal statements returns, while for the second condition the while statement sends the execution message to its next node if there is any.

• **If statement** has a syntax like if (exp) cmd_stat1 else cmd_stat2. The node constructs are shown in Figure 5.4c. The execution is very simple. First it executes the exp node. If the return value is true it executes the cmd_stat1 node; otherwise, cmd_stat2. For statement has a syntax like for (exp1; exp2; exp3) cmd_stats. The corresponding node constructs are shown in Figure 5.4d. These nodes do actions similar to other statement nodes.

• **Switch statement** has a syntax like

```java
switch (exp) {
    case exp1: cmd_stats1;
    case exp2: cmd_stats2;
    ...
    default: cmd_stat_default;
}
```

The switch statement executes the exp node and stores the returned parse value. Then it sequentially walks through each case statement. For each case statement, it
executes the corresponding case-expression and compares the two values. If the two values are equal, it then executes the case-command statements. Of course, the execution of any terminal command statements will change the behavior. The node constructs are shown in Figure 5.4e.

- **Other statements** include file, db, and p21 statement. Each of these nodes contains a contents node sub tree for their argument expressions.

### 5.3.3 Types of Expression Nodes

- **Unary operators** refer to operators, such as +, -, !, ++, --, etc. They are all constructed as two connected nodes—an operator node and an operand node. The operand node is the contents node of the operator node. The next node is always null. For ++ and -- (both pre and post operators), the operator node first gets the symbol value of its contents, then performs the increment or decrement on its contents node, and at last returns a proper value. For all other operators, each operator node executes its contents node and get the returning value, then returns the value after the proper operation.

- **Binary left operators** are those which work on two operands, such as +, -, *, etc. Each construct has three nodes—the left operand is the contents of the operator node and the right operand is the next node. Each operator node executes both its contents and next nodes, makes the arithmetic operations on the two values returned from the two executions and finally returns the value.
Figure 5.4: Some statement node constructs
- **Binary right operators** are those which work on two operands, such as `+=`, `-=`. `*=` etc. Each construct has three nodes—the left operand is the contents of the operator node and the right operand is the next node.

- **Constants** are represented by only one type of nodes (node type `exp_const`). The constant is stored in its `val` attribute.

- **Symbols** can be a variable name or something in the format like `$exp$. The former can be represented by only a single node called `exp_symb`. The latter can be represented by two nodes—an `exp_symb` node and an expression node pointed by its next pointer. Symbol `$exp[exp]$` has three nodes. In addition to the two nodes seen in `$exp`, the third node is an index expression node pointed by the contents pointer of the `exp_symb`. This nodes returns a symbol value, which resolved from the corresponding `SYMBOL_NODE` and an optional index value, which is stored in the contents node of the symbol node.

- **Procedure calls** can be achieved in two ways. The first way is directly using the procedure name. The second way is in `$exp$` format, where `exp` has a value of a procedure's name. Both methods can have a list of mixtures of expressions, arrays, reference to arrays or scalar variables. The different types of arguments have been presented using different node types, such as `exp_fn_arg_ref_scalar`, `exp_fn_arg_val_array`, `exp_fn_arg_ref_array`, and `exp_fn_arg_ref_scalar`. Figure 5.5 shows the node constructs for `$exp$` method. The process of calling a procedure is explained fully in Section 5.5.7.
5.3.4 Typical Function Nodes

Tnr provides many built-in functions. Each function consists of a function name token and zero or more arguments. For a function with no arguments, the node construct is simply one function node (for example, the node constructs for function `last_rexp()` consists of only one node of type `fn_last_rexp`). Functions with one argument have at least three nodes—the function node, the function parameter node (type name `fn_param`), and the expression node for that parameter. The `fn_param` node is the contents of the function node and the expression node is the contents of the `fn_param` node. For functions with more parameters, the node constructs are similar. The nodes for the second or later parameters are pointed by the next pointer of the former one. Figure 5.6 shows a general node constructs for a function call. When the function is executed, each expression is executed and the value is stored in its `fn_param` node. The `fn_xx` node then performs the recursive operations on these values.
5.3.5 Select Rule Nodes

Figure 5.7 shows the constructs of select_rule() function. The node constructs start with a node of type fn_select_rule, and this node has no node on its next side. Its contents is a list of fn_param nodes. As each argument of this function is a rule procedure, the fn_param node has contents as procedure calls, which have been sketched in Figure 5.5. When fn_select_rule node executes, each exp_fn_call node is executed until one of the execution returns "matched" value. (Section 5.5.7 fully explains how an exp_fn_call node executes itself.)

5.4 Significant Global Variables

- **Code_file_stack** is a stack storing all the Tnr programs specified in the command line or included by include statement in Tnr programs.
- **Delimiter** is a delimiter string specified by delim() and used by merge(), split() and other functions.
- **Err_file** is the file stream pointer pointing to the current error device.
Figure 5.7 Node construct for select_rule() function

- **Global_var** is a SYMBOL_NODE list of global variables.
- **In_file** is the file stream pointer pointing to the current input file.
- **Is_rule_matched** is a flag indicating whether the called rule procedure is matched or not.
- **Matching_string** holds the last matching string value which occurs in move(), select_rexp(), etc.
- **Out_file** is the file stream pointer pointing to the current output file.
- **Select_rexp_delimiter** is the normal string which decides the input unit for the select_rexp() function.
- **Template_root** is the root of the SYMBOL_NODE tree. A Tnr program's node tree starts from it.
- **TNR_Error_detected** is a flag indicating whether a Tnr error has occurred.
5.5 Implementation Algorithms and Major Routines

This section details some major algorithms and routines applied in the implementation of the Tnr interpreter.

5.5.1 Manipulation of Symbol Nodes

- **add_var_param()** adds a new symbol node to the global variable node list, a procedure's parameter list, or a procedure's local variable list depending on the type of the symbol node.

- **add_symbol()** creates a new symbol node based on the given type and name string and returns the handle to the node.

- **assign_symbol_default()** assigns a symbol node a default parse value. The symbol node is not dereferenced.

- **assign_symbol_value()** assigns a symbol node a specified parse value. If the symbol node has a reference, the value is assigned to the referenced node. This process may continue recursively.

- **find_symbol()** searches for a symbol node based on its name in a SYMBOL_NODE list. If a node is found, the handle to the node is returned; otherwise, a NULL is returned.

- **find_symbol_node()** is a helper-function called by both assign_symbol_value() and resolve_mat_symbol(). It accepts a parse node pointer pointing to the node constructs of $exp$ as input and resolves the destination symbol node referenced by that node constructs.
find_var_param() first looks for the symbol node in the specified procedure's parameter and variable tables according to the given symbol name (the procedure is specified by the handle to the procedure's symbol node). If a node is found, its handle is returned; otherwise a NULL is returned.

get_symbol_node() resolves the corresponding symbol node for a given parse node.

malloc_symbol_default() increases a given symbol node's internal SYMBOL_VAL array size. The existing values are retained. This function is called by Yacc code to allocate memory for variable declarations.

resolve_mat_symbol() returns the destination variable pointer for a given parse node. This is a general interface for resolving the matching variable based on a given PARSE_NODE. Of course, the parse node must be either a variable parse node or a node construct for expression of $exp.

5.5.2 Manipulation of Variables

assign_variable_value() assigns a parse value to a specified variable's specified element. If the index is greater than the greatest index plus one, the gap will be filled with elements with uninitialized values.

clean_variable() frees all the memory occupied by the internal symbol value array for the given variable.

malloc_variable_value() increases a variable's internal array size for holding more symbol values. The existing value is retained.
5.5.3 Manipulation of Parse Nodes

- **add_node()** creates a parse node based on the given type. The contents and next nodes are also specified at the creation.

- **append_node()** appends a parse node to another as the next-node.

- **append_contents()** appends a parse node to another as the contents node.

- **exec_node()** is the heart of the Tnr interpreter. It executes the whole parse node tree. The execution starts from the root parse node and the execution message propagates among the nodes automatically.

5.5.4 Manipulations of Multiple Tnr Programs

A programmer can specify multiple source code files in two ways—specifying multiple files in command line or using the include keyword in a Tnr program. The interpreter manages the multiple source code files using a “file entry stack”. A file entry includes information, such as the file index, the file stream pointer, and the current input line number. Utilities for supporting the management of multiple source code files include:

- **add_code_file_list()** inserts a new source file entry into the internal source code file list and returns an index to that file. If the file is already scanned, -1 is directly returned without further action.

- **enqueue_code_file()** appends a file entry to the bottom of the source code file stack.

- **push_code_file_stack()** pushes a file entry into the stack. pop_code_file_stack() pops up the top file entry in the stack. get_file_name_by_index() retrieves the file name according to its index.
### 5.5.5 File Stacks

As stated in Chapter 4, Tnr does not need programmers to keep track of the handles of input/output files, database connections, and STEP databases. Instead, Tnr automatically manages them by the mechanism called file stacks.

Internally Tnr owns an global variable called File_db_list, a pointer to a stack, whose element is a data structure called FILE_DB_LIST shown below:

```c
typedef struct file_db_list FILE_DB_LIST;
struct file_db_list {
    int type;
    char* name;
    FILE* fd;
    char* db_link;
    FILE_DB_LIST* next;
};
```

*Type* can be one of the following enumerated values: SOCKET_RD, SOCKET_WR, PIPE_RD, PIPE_WR, FILE_WR, FILE_RD, DB_ORACLE, and P21_FILE (at the time of writing this dissertation, part 21 files have been changed to be managed by EQL. Therefore, P21_FILE is obsolete). These type names are quite self-explanatory, so we do not describe them here. The attribute name together with type serves as the I.D. for the stack elements. It is equal to a file name for a file or a command string for a pipe. For a socket, it equals to a string of format port: host, where port and host are the port number and the host name of the socket, respectively. For an Oracle database, its value is a string of format oracle_db_link_n, where n is a unique integer. *Fd* is the pointer to the proper file stream for proper types of data resources (for database connection this field is null). *Db_link* keeps track of (Oracle) database name.
5.5.6 I/O functions

The current implementation of I/O functions uses its own state machine to parse the format string. The state machines are embodied by two tables—scanf_trans_matrix and printf_trans_matrix. The implementation has a special module for supporting I/O called tnrio.c. The module has two functions return format sets sequentially (one for input and the other one for output). The I/O functions consecutively call these two functions, respectively, to get the next format set. Then according to the last character of the set it reads or prints.

Maybe the most difficult task in the implementation of input functions is the memory management. For example, if the input unit is a string, it is impossible to know the required memory size beforehand. The current implementation needs to read files twice in such cases. The first time it gets the size and then seeks back what just has been read using fseek function. The second time it does the real read and assignment. This solution limits the input functions so they cannot be applied on input streams. In future implementation, we can attach to each input file buffer. The input functions and other related functions, such as select_rexp(), can read from the buffer instead of from the file stream directly. The buffer is in charge of the input from a file. A buffer has two goals: it should make sure that input functions can safely roll back any reasonable number of characters it has read and, at the same time, it does not sacrifice too much memory.
5.5.7 Procedure Calls

A procedure can be called in two ways. One way is to use procedure-call statements directly, while the second way is by function select_rule(). The later case requires all the calling procedures to be rules.

The tasks supporting procedure calls can be divided into several steps. The first step is to match the parameter table of the called procedure and the arguments provided by the calling statement. If there is any data type mismatch (scalar and array) or the number of arguments is greater than that for parameters, a runtime error is reported. The second step is to push the function environment stack, including the current input and output files; the current database connection, the current STEP database connection, and the current parse node.

The third step is to create a fresh set of variables (using the VARIABLE data structure) for all parameters and local variables in the called procedure and assign each variable its default value. These variables are temporarily stored on the new_val attribute of the SYMBOL_NODE, respectively.

The fourth step has different actions to take upon different argument passing methods: to call by value or call by reference. If an argument is called by value, Tnr needs to copy the SYMBOL_VAL(s) (one or more values depending on the value type) which are held by the final dereferenced SYMBOL_NODE to the SYMBOL_NODE of the corresponding function parameter. If it is called by reference, Tnr needs to set the proper linkage for the parameter SYMBOL_NODE to the final dereferenced SYMBOL_NODE inside the calling procedure. If the reference is an array element, Tnr also needs to store the element index to the parameter SYMBOL_NODE (in field
var_index). Figure 5.8 summarizes all the different scenarios. After that, Tnr pushes the variables pointed to by new_val into the stack pointed by val to become the current values of the variables/parameters used by the function. ew_val is then set to NULL.

Now that all the preparations have been finished, the PARSE_NODEs pointed by parse_tree can be executed. After the execution, each stack of the SYMBOL_NODE pops up the top of the variable stack (pointed to by val) to reclaim occupied memory to the operating system.

Until now, the only issue left which we have not discussed is how to return a value from a called procedure. The returned value is stored in a field of function environment data structure. After the called procedure is finished, the calling procedure can retrieve the return value from its function environment.

5.5.8 Interface to SQL

Tnr does not try to parse the SQL statement by itself to support SQL host variables; instead, it lets the SQL server do it (the current implementation only supports Oracle SQL server). Oracle SQL server describes the information about bind-variables upon request after receiving a SQL statement. The information includes the number of bind-variables and each variable's name. Based on the name of each variable, Tnr can search the variable from both the parameter table and variable table of the local function or the global variable table. If the searching fails or the variable is not a proper type (i.e., in array data type), a runtime error is reported; otherwise, Tnr retrieves the value of a variable and writes it into the memory of the Oracle bind variable descriptors. (For more information about Oracle bind variable descriptors, please see reference[31].)
<table>
<thead>
<tr>
<th>case</th>
<th>The last dereferenced symbol node in the calling procedure</th>
<th>Parameter in the called procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>f(a) or f(a[])</td>
<td>copy each symbol value inside the left array to the right.</td>
<td></td>
</tr>
<tr>
<td>f(a[i])</td>
<td>copy the ith symbol value inside the left array to the right.</td>
<td></td>
</tr>
<tr>
<td>f(&amp;a) or f(&amp;a[])</td>
<td>make the var of the right VARIABLE node point to the left one.</td>
<td></td>
</tr>
<tr>
<td>f(&amp;a[i])</td>
<td>make the var of the right VARIABLE node point to the left one and right node saves the index (var_index = i).</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.8: Procedure call
5.5.9 Interface to Eql

In Chapter 3, we mentioned the EQL API utilities. These utilities have been deployed to build up the interface between Tnr and Eql. First Tnr uses EQL API function `eql_parse()` to parse a query string. The function returns a list of host variables, if any, for the query. After Tnr successfully resolves the values for these host variables, it calls the function `eql_bind_host_variable()`, where each host variable is bound by the value of its Tnr peer. Last, Tnr calls for service of `eql_process()` where the query is executed. The function returns the query type; in addition, it sets up the query results and the result array size.

As with the SQL interface, arrays are not allowed to be used as host variables while each individual element is permitted. The value binding involves the following intelligent data conversions on the Tnr side. If a variable is of string data type and the value looks like a logical value, such as “.U.” or “.F.”, then convert the value into a logical one. If a variable is of integer or real number, convert into a numerical value. If a variable looks like an enumeration value—i.e., characters surrounded by two dots and together surrounded by two single quotes—it converts into an enumeration number. If a variable’s string value looks like an instance value (# followed by an integer), it converts into instance; otherwise, it assigns the string value directly into the host node.
Chapter 6

Software Integration Using Tnr and Eql

6.1 Introduction

Chapters 2 and 4 introduced the syntax of Eql and Tnr, respectively, and Chapters 3 and 5 discussed the implementation issues of Eql and Tnr, including internal data structures, major algorithms, etc. This chapter presents two examples to demonstrate the application of Eql and Tnr. The first example is the follow-up of the university database example presented in Chapter 2. We will develop two programs to migrate data from the STEP database to an Oracle database and to generate two simple reports based on the STEP database—a class catalog and a student grade report. The second example is the Intelligent Manufacturing Workstation (IMW) project in the Center for Advanced Software System Integration (CASSI) at Ohio University. This project has extensively applied Eql and Tnr in its system integration tasks.

6.2 Example One

Figure 6.1 shows the data flow directions in our example. Other directions can be implemented with similar methods. For the data translation, a relational database
schema has been designed and is shown below. Notice that for the simplicity reasons, the relational database schema is not the exact mapping of the EXPRESS

Figure 6.1 Data migration using Tnr programs
Table class_offering
  title: VARCHAR(20) NOT NULL,
  id: VARCHAR(20) NOT NULL,
  callNumber: VARCHAR(20) NOT NULL,
  theQuarter: VARCHAR(6),
  year: INTEGER,
  theInstructor: CHAR(9),
  theAssistant: CHAR(9),
  assignmentWeight: REAL;
  examWeight: REAL;
  Key (id);

Table grade_work
  name: VARCHAR(20) NOT NULL,
  description: VARCHAR(200),
  possibleScore: REAL NOT NULL,
  weight: REAL NOT NULL,
  theClass: VARCHAR(20) NOT NULL,
  type: VARCHAR(4) NOT NULL,
  Key (theclass, name);

Table grades
  Class: VARCHAR(20) NOT NULL,
  Name: VARCHAR(20) NOT NULL,
  Student: CHAR(9) NOT NULL,
  Grade: REAL,
  key (class, student);

Table people
  firstName: VARCHAR(20) NOT NULL,
  middleName: VARCHAR(20),
  lastName: VARCHAR(20) NOT NULL,
  id: CHAR(9) NOT NULL,
  position: VARCHAR(5) NOT NULL,
  Key (id, position);

Table enrollment
  student: CHAR(9) NOT NULL,
  class: VARCHAR(20) NOT NULL,
  courseGrade: REAL,
  Key (student, class);
6.2.1 Data Translation From STEP To Oracle

The mission is to transfer all the related data from the STEP database into an Oracle database. We assume that a database which contains all the tables already exists on an Oracle database space and a SQL server is also running. The following steps can be used to achieve our goal:

- For each Oracle table, design one or more Eql queries which can be used to extract all the required data by that table.
- Construct a SQL insertion queries for each table. Introducing host variables may make the queries independent of the data and Tnr program.
- Use `eql_exec()` function to execute those queries of Step 1 and retrieve all the data out of STEP database.
- Use `sql_exec()` to carry out the SQL queries.

The Tnr code is given below.

```
// File name: migration.tnr
// Date: 08/02/98
// Author: Lihong Huang
// Description:
// This program translates data from a STEP database to
// a Oracle database. The relation schema has be created
// based on the EXPRESS schema before hand.
// Usage:  tnr -f migration.tnr -v <step_db> -v <ora_db>

include "sql.queries"
include "eql.queries"
```
template main(step_db, ora_db)
var data[];
{
    if (!is_init(step_db) || !is_init(ora_db)) {
        error("usage tnr -f migration.tnr -v <step_db> -v <ora_db>
    exit;
    }
    p21(step_db);
    db("ORACLE", ora_db);

    transfer_data(Get_class_offering, Put_class_offering);
    data[0] = "HW";
    transfer_data(Get_assignment, Put_grade_work, 5, data[]);
    data[0] = "EXAM";
    transfer_data(Get_exam, Put_grade_work, 5, data[]);
    data[0] = "INST";
    transfer_data(Get_instructor, Put_people, 4, data[]);
    data[0] = "STD";
    transfer_data(Get_student, Put_people, 4, data[]);
    transfer_data(Get_enrollment, Put_enrollment);
    transfer_data(Get_assignment_grade, Put_grade);
    transfer_data(Get_exam_grade, Put_grade);
}

The first two lines use include keyword to input two files—sql.queries and eql.queries, respectively. As we have stated, one should carefully design the Eql and SQL queries based on the EXPRESS and relation schemas during the early stage of the
data migration. These queries can be placed into separate files so that they are independent of any Tnr programs. If there are any changes to the schemas, one only need to modify the queries. The Eql queries are given in the following list.

```
// All Eql queries for this example
// (stored in file eql.queries).

var

Get_class_offering = "select (title, id, callNumber, theQuarter, 
  year, theInstructor.id, theAssistant.id, assignmentWeight, 
  examWeight) from CLASS_OFFERING";

Get_assignment = "select (name, description, possibleScore, 
  weight, theClass.id) from ASSIGNMENT";

Get_exam = "select (name, description, possibleScore, weight, 
  theClass.id) from EXAM";

Get_instructor = "select (firstName, middleName, lastName, id) 
  from INSTRUCTOR";

Get_student = "select (firstName, middleName, lastName, id) 
  from STUDENT";

Get_enrollment = "select (theStudent.id, theClass.id, courseGrade) 
  from ENROLLMENT";

Get_assignment_grade = "select x->theClass.id, y->theWork.name, 
  x->theStudent.id, y->grade from x in enrollment, 
  y in x->assignmentGrades[]";

Get_exam_grade = "select x->theClass.id, y->theWork.name, 
  x->theStudent.id, y->grade from x in enrollment, 
  y in x->examGrades[]";
```
Get_catalog_info = "select (title, id, callNumber, theQuarter, 
    year, theInstructor.firstName, theInstructor.middleName, 
    theInstructor.lastName, theAssistant.firstName, 
    theAssistant.middleName, theAssistant.lastName) 
from class_offering";

Get_transcript_info = "select s, s(firstName, middleName, 
    lastName), e->theClass(callNumber, title, gradeMap), 
    e->courseGrade from s in student, e in enrollment 
where s == e->theStudent";

The SQL queries are given in the following list.

// All SQL queries for this example
// (stored in file sql.queries).

var

Put_class_offering = "insert into CLASS_OFFERING values ((title, 
    id, callNumber, theQuarter, year, theInstructor, theAssistant, 
    assignmentWeight, examWeight) = 
    (:r[0], :r[1], :r[2], :r[3], :r[4], :r[5], :r[6], :r[7], :r[8]))";

Put_grade_work = "insert into GRADE_WORK values ((name, 
    description, possibleScore, weight, theClass, type) = 
    (:r[0], :r[1], :r[2], :r[3], :r[4], :r[5]))";

Put_people = "insert into PEOPLE values ((firstName, middleName, 
    lastName, id, position) = (:r[0], :r[1], :r[2], :r[3], :r[4]))";

Put_enrollment = "insert into ENROLLMENT values ((student, 
    class, courseGrade) = (:r[0], :r[1], :r[2]))";

Put_grade = "insert into GRADE values ((class, name, student, 
    grade) = (:r[0], :r[1], :r[2], :r[3]))";

Another issue to pay attention to is taking full advantage of Eql. The powerfulness of Eql can be easily underestimated. For example, as Eql provides very
powerful loop functionalities, one should shift the for-loop requirements from Tnr side to Eql whenever it is possible. In this way, a program is both more efficient and more manageable.

Coming up in the program is the main procedure. First we need to open the STEP database and connect to the Oracle server using p21() and db() statements respectively. After that, for each table, we pick up the proper Eql and SQL queries for each table and carry out these queries using eql_exec() and sql_exec() functions. The retrieved data from the STEP database are stored inside array data[]. Each element of the array stores the information corresponding to each row of the SQL table, therefore, we need to process the array by split each element into another array so that the SQL queries can use (the new array name should match what is expected by the SQL query, in this case, is r[]). All these tasks are fulfilled by procedure transfer_data. Sometimes, additional information which can not be found in the STEP database needs to be provided (for example, when two different entities are merged into one table, we may need a column indicating the entity type, whose value can not be directly extracted from a STEP database and need to be provided by the programmer).

6.6.2 Report Generation

In the second portion of this example, we use Tnr to generate two reports: one is the class catalog and the other is the student grade report. Eql has been applied to extract data from the STEP database.
// File name: catalog.tnr
// Date: 07/26/98
// Author: Lizhong Huang
// Description:
// This program extracts information of all classes
// offered and generates a human readable report.
// Usage: tnr -f catalog.tnr -v <db_name> -v <report_name>

#include "eql.queries"

template
template

main(stepDB, report)
var i; result[]; row[]; name;
{
    if (!is_init(stepDB) || !is_init(report)) {
        error("usage tnr -f catalog.tnr -v <step_db> -v <report_name>
            "n");
        exit;
    }
    p21(stepDB);
    eql_exec(Get_catalog_info, result[]);
    file(report);
    for (i=0; i<#result[]; i++) {
        split(result[i], row[]);
        printf("%-20s: %s
", "Name", row[0]);
        printf("%-20s: %s
", "ID", row[1]);
        printf("%-20s: %s
", "Call Number", row[2]);
        printf("%-20s: %s
", "Quarter", substr(row[3], 2,
                length(row[3])-2));
        printf("%-20s: %s
", "Year", row[4]);
        merged(name, " ", row[5], row[6], row[7]);
        gsub("\[\] \*", "", name);
        if (length(name) == 0)
            name = "N/A";
        printf("%-20s: %s
", "Instructor", name);
        merged(name, " ", row[8], row[9], row[10]);
        gsub("\[\] \*", "", name);
        if (length(name) == 0)
            name = "N/A";
        printf("%-20s: %s
", "Teaching Assistant", name);
The example only needs to apply an Eql query to extract data out of the STEP database, then it generates a formatted text file—a class catalog using printf() functions. Most of the code are straightforward. The only complexity is the calculation of a person's name. In the STEP database a person's first, middle, and last name are stored in three separate attributes, some or all of the attributes may have “unknown” value for the given query. Obviously it is not proper to use $ directly inside a person's name. So the code first concatenates the names in order using a white space as the delimiter, then it removes all the instances of $ and following white spaces. The result string is the full name in good format. If the result string is empty (length is equal to zero) then it prints out “N/A”.

Name : Graduate Coloquim  
ID : 12345  
Call Number : EE690  
Quarter : WINTER  
Year : 1998  
Instructor : N/A  
Teaching Assistant : N/A

Name : Advanced CIM  
ID : 12345  
Call Number : EE690  
Quarter : WINTER  
Year : 1998  
Instructor : Bob P Judd  
Teaching Assistant : N/A

Name : Programming With C++  
ID : 67890  
Call Number : EE590  
Quarter : FALL  
Year : 1997  
Instructor : Bob P Judd  
Teaching Assistant : N/A
A student grade report can be treated in a similar way, except the Eql queries are a little more complex. Notice that the internal object I.D. of STEP database has been applied to distinguish the different data groups. We list the code and result here without much explanation.

///
/// File name: transcript.tnr
/// Date: 07/26/98
/// Author: Lihong Huang
/// Description:
/// This program generates a formatted grade report
/// for all students based on the given STEP database.
/// Usage: tnr -f transcript.tnr -v <step_db> -v <report_name>
///
///
include "eql.queries"

template
main(step_db, report)
var result[], i, r[]; student_oid = "#0"; name;
{
  if (!is_init(step_db) || !is_init(report)) {
    error("usage trn -f transcript.trn -v <step_db> -v <report>", 
      "n");
    exit;
  }
p21(step_db);
eql_exec(Get_transcript_info, result[]);

  file(report);
  for (i = 0; i < #result[]; i++)
  {
    split(result[i], r[]);
    if (strcmp(student_oid, r[0]) != 0)
    {
      student_oid = r[0];
      merged(name, " ", r[1], r[2], r[3]);
      gsub("[$] [ ] ", ", name);
      printf("%nStudent: %s
", name);
      printf(" %s%-%s%-%s%-%s%-%s
", "Call Number", "Title", "Grade");
      printf("%s%-%s%-%s%-%s
", "-------------------", "-----");
    }
    printf("%s%-%s%-%s%-%s
", r[4], r[5], grade_map(r[6], r[7]));
  }
}

rule
grade_map(gradeMapStr, grade)
var r[];
{
  if (strcmp(gradeMapStr, "$") == 0)
    return ""
  splitd(substr(gradeMapStr, 2, length(gradeMapStr)-2), ",", r[]);
  switch (1)
  {
    case grade >= r[10]: return "A";
    case grade >= r[9]: return "A-";
    case grade >= r[8]: return "B+";
  }
}
```csharp
case grade >= r[7]: return "B";
case grade >= r[6]: return "B-";
case grade >= r[5]: return "C+";
case grade >= r[4]: return "C";
case grade >= r[3]: return "C-";
case grade >= r[2]: return "D+";
case grade >= r[1]: return "D";
case grade >= r[0]: return "D-";
default: return "F";
}
)
```

The result is shown below.

<table>
<thead>
<tr>
<th>Student: Julia R Lee</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Number</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>EE690</td>
</tr>
<tr>
<td>EE590</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student: Ema R Lee</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Number</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>EE590</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student: Kevin L Smith</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Number</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>ET855</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student: Dorothy A King</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Number</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>ISE490</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student: Tom B Tang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Number</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>ISE490</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student: Larry Maher</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Number</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>EE690</td>
</tr>
</tbody>
</table>
6.3 Example Two

Both Eql and Tnr have been extensively applied in the software integration activities in the Center for Advanced Software System Integration (CASSI) at Ohio University. One very interesting project is called Intelligent Manufacturing Workstation (IMW). It applies both Eql and Tnr to integrate several state-of-the-art commercial software systems together to create an integrated manufacturing designing system. Part of the work has appeared in a thesis by Subramanian[39] et al at Ohio University.

6.3.1 IMW & IMDE

The IMW is very closely related to the Integrated Manufacturing Design Environment (IMDE). IMDE is a virtual manufacturing designing environment specification, and the IMW is a specific application which conforms to the specification of IMDE. The goals of IMW are listed below.

1. Plan machining operations
2. Select toolings
3. Determine feeds and speeds
4. Develop cutter paths
5. Calculate execution times
6. Develop cost estimates

7. Develop schedules and critical path

8. Create machine programs

9. Verify program

The commercially available tools to be integrated are a computer-aided process planning system (PART), a 3D simulation system (VNC), and a cost evaluation system (CA). The architecture of IMW is illustrated in Figure 1.1.

6.3.2 UDMM

The UDMM (Unified Design Meta Model) is the core of IMDE. All data types in IMW can be found in UDMM. UDMM has been implemented using EXPRESS. All entities in UDMM EXPRESS schema can be classified into three categories: independent entities, dependent entities, and utility entities. Independent entities model the information pertaining to objects that exist in the real world and identifiable by some particular key. Dependent entities have to uniqueness characteristics associated with them. Utility entities are really dependent entities which have been used within multiple independent entities. Independent entities include WorkstationClass, MachineProgram, OperatorClass, TooledListWorkstation, WorkCellClass, PortableToolClass, ToolAssembly, ToolElement, PartAssembly, PartElement, Feature, Operation, and Setup. Dependent entities include FeatureDescription, FeatureParameter, and MachinedFeatureLink. Utility entities include Parameter, IntegerParameter, TextParameter, RealParameter, CurrencyParameter, USDollar, Speed, RevPerSec,
6.3.3 Implementation of IMW Using Tnr and Eql

The goals of the implementation are to provide data flow from PART to CA, PART to VNC, UG to PART, UG to CA, and UG to VNC. These translators are comprehensive enough to demonstrate the most complex techniques required to implement the full-featured data translators.

- **PART Translator**: The mission of PART translators is to extract domain information from PART system and create STEP databases. The challenges faced by part translators have introduced some interesting methods to programs using Tnr. One of the challenges is to dispatch importing requests, which comes up from the anticipation of the future addition of rules to export. With a deliberate naming convention and interface specification for the rules, a Tnr program is capable of dynamically resolving appropriate templates or rules. Another challenge is the interrogation of the PART database. The difficulty is how to isolate the future changes of the database structures from the current implementation of the translator. The solution has been invented by the author of the translator by gathering all queries into a file (as in this case, no dynamic data resolution is needed) and initializing the relevant global variables using the information stores in the queries file. If the database structures are changed, only the queries inside the file need to be replaced by proper ones accordingly.
• **VNC Translator:** The VNC translator reads a STEP database and creates a hierarchy of directories and files usable by VNC and allows a setup to be visualized in the VNC environment[14]. VNC libraries consist of files organized into a hierarchy of directories (the path information is stored in .vncpthfig file which, is read at the startup of VNC). The IMW defines a standard library hierarchy and VNC allows some variation in the hierarchies. Most of the work for the VNC translator is to generate CLI (Command Line Interface) files, which are executed by VNC. The IMW must place them in the right location.

• **CA Translator:** The CA translator reads a STEP database and generates a set of CA system files for the cost analysis. One way for CA to import data is from design command files. The data translator creates these commands based on the STEP database generated by the PART translator. UG translators applied similar techniques and are not discussed here.
Chapter 7

Conclusions

Software integration based on the data translation method will get more attractive with language level programming support. This is because such a language will reduce the too overwhelming efforts needed for data translator implementation using system languages. In this work, two new languages—Eql and Tnr—have been designed and implemented on both IRIX and HP-UX platforms. Eql, an SQL-structure-like data manipulation language (DML) on STEP (ISO 10303) data, it provides powerful syntax and highly efficient mechanisms for aggregate operation and comparison, inverse relationship resolution, and other advanced features of object oriented database manipulation. Tnr unifies many features of other languages, such as Awk, Lex, Yacc, and interactive shells, and provides unified interfaces at the language level to data resources, such as ASCII files, relational databases, STEP databases, pipes, sockets, etc.

Some applications of these languages, especially the implementation of IMW at the Center for Advanced Software System Integration at Ohio University, are also presented and discussed to demonstrate the applicability of the languages.
The author anticipates that Tnr and Eql, as other programming languages, will experience the evolution processes as they mature. The evolutions will much depend on the programming practice inside the user communities. However, there are some foreseeable candidates for future research at the time of writing this dissertation.

7.1 EXPRESS Query Language

For Eql, the more accurate performance of SDAI library is essential for the significant improvement of Eql. Because the SDAI specifications have not yet reached a stable stage, the author anticipates that Eql implementation codes may need to be updated to be compatible with the future SDAI libraries. However, the changes should be minor.

Another issue is related to the philosophy for the creation of Eql; that is, its usefulness rather than completeness. Completeness theory about object oriented data manipulation languages has not yet been found, and Eql has no intention to be complete. As an example, select results sorting and aggregate value calculations (maximum, minimum, average, etc.) are not supported by the current definition and implementation of Eql. It is the author's belief that Eql should, most of the time, be used together with Tnr. While Tnr provides similar calculation abilities, Eql should not repeat the efforts. However, in the future if users find it is useful for Eql to have these features by itself (e.g., if Eql is used in interactive mode) Eql should be furnished with these features.
7.2 Templates and Rules

For Tnr, there are two issues under consideration. The first is the limitation of scan and other data input functions. As we discussed before, these functions, in the current implementations, cannot work on stream input devices properly. The future implementation can be based on buffered data input. Each data input resource is attached to a buffer, and the reading activities can be layered in three levels. The first level are the data input functions, the second level is the data buffer, and the bottom layer is the stream input device. The input functions use several services provided by the buffer to move the reading position back and forth. Because the functions know when and how much of the data in the buffer can be discarded, they can use the services (functions) provided by the buffer to do so. However, it is the buffer's responsibility to know when there is a need to refill data from the input devices. By this simple mechanism, the input functions never need to worry about the input stream manipulations.

The second issue is the further study of the rule-expressions. The current regular expressions, such as move(), select_rexp(), etc. have been found very useful. However, select_rule() has not been found to be as useful as the others. A text file (especially a text data file) usually follows some certain format rules, and when a programmer extracts information out of it, he or she wants to specify these rules. The goal of select_rule() is to provide Tnr a "grammar specification" ability. One may have realized that select_rexp() works like Lex—it tries to recognize string tokens using regular expressions. However it would be too overwhelming to use regular expressions alone to
express a grammar rule. We need something like Yacc to cast rules, and ideally
select_rule() should play this role well.

Other issues, such as supporting associative array, separation procedures into
rule and template, etc., should also be measured within the next few years. It is too early
for the author to make a judgment now.
Bibliography


37. Stevens, Ron, Redman, Jim., and Cooper Elizabeth. “Plug and play component
software for manufacturing and control.”


38. Stylianou, A. C. and Jefferies, C. J. and Robbins, S. S. “Corporate Mergers and the

39. Subramanian, Chandrasekar. The Intelligent Manufacturing Workstation: A Partial

40. Thomas, Anne. “Enterprise JavaBeans Server Component Model for Java.”


41. Vinoski, Steve. “CORBA: Integrating Diverse Applications Within Distributed
Heterogenous Environments.” IEEE Communications Magazine. 14(2), February,
1997

42. Wall, Larry and Schwartz, Randal L. Programming Perl. O'Reilly & Associates,
Inc., Sebastopol, California 95472, 1992