Chapter 4

Implementation

This chapter goes into low-level detail about the implementation of the Squeal interpreter. It is likely to be of interest to anyone modifying the Squeal interpreter or building a similar system. Readers who are primarily interested in the conceptual ideas of the thesis or in using the system may want to skip to the next chapter, which discusses applications.

The Squeal interpreter is written in Java and runs on a Sparc-20/50 workstation. It communicates through a network using TCP/IP with an Intel-based machine running Microsoft SQL Server. This chapter describes the classes in the Java implementation of Squeal. Outside tools and classes used are:

- Java Compiler Compiler [36], discussed below
- The OROMatcher classes for Perl5 regular expressions [29]
- The HtmlStreamTokenizer class for parsing HTML [8]
- Santeri Paavolainen's implementation of MD5 [30, 34]

To distinguish standard Java classes from those defined for Squeal, Squeal class names are underlined.

4.1 Database state

The class DBstate is used to encapsulate database state. When Squeal is started, a connection to the SQL server is opened and an instance of DBstate is created. The connection is closed when Squeal is terminated. DBstate has one public member variable, **stmt1**, which is of type java.sql.Statement and is used to send a query or update to the SQL server.

4.2 Column

The column class is used to represent each column in tables on the SQL server. The class is needed in interpreting SELECT statements and in displaying results. Member variables (Figure 4-1 hold each column's name and type as well as the Table instance (defined below) of which they are part.

4.3 Tables

4.3.1 Purpose

In addition to the SQL tables stored on the SQL server, the Squeal implementation keeps information locally about every table accessible to the user or system. This is used for interpreting SELECT statements and when user tables are created, deleted, or modified. Every table is an instance of the Table class. Figure 4-2 shows the hierarchy beneath Table, including instance variables. All of

`public String name`
The name of the column
`public String type`
The type of the column
`public Table table`
The Table instance of which this is a component

Figure 4-1: Member Variables for Column

Table
- **computation**
- **creation**
- UserVisibleTable
  - AutomaticTable (section 3.5.2)
    - **link** (section 2.2.3)
    - **page** (section 2.2.1)
    - **parse** (section 2.2.1)
    - **rcontains** (section 2.2.4)
    - **rlink** (section 2.2.4)
    - **tag** (section 2.2.2)
  - DerivedTable (section 3.5.3)
    - **att** (section 2.2.2)
    - **header** (section 2.2.3)
    - **list** (section 2.2.3)
  - UserReadableTable (section 3.5.1)
    - **urls** (section 2.2.1)
    - **valstring** (section 2.2.1)
    - UserDefinedTable

Figure 4-2: Class Hierarchy of Tables

`static Hashtable tables`

Hash table mapping table names to Table instances

`static Vector url_id_columns`

The set of all columns defined in the system of type **url_id** (section 2.2.1)

`public String name`

The name associated with the Table instance

`public Vector columns`

The set of Column instances associated with the Table instance

Figure 4-3: Member Variables for Table

creation		
colname	*type*	*notes*
tab	VARCHAR(15)	primary key
stamp	DATETIME	
def_value_id	INT	

Table 4.1: The **creation** relation

the tables discussed up to now have been subclasses of UserVisibleTable. These include the tables defined in the ontology as well as any tables created by the Squeal user. The Squeal internal tables, which are not subclasses of UserVisibleTable, are discussed later in this section.

4.3.2 Details

There are two static variables associated with Table:

1. *tables*, a HashTable that maps strings to the Table with that name
2. *url_id_columns*, a Vector containing the instances of Column that have type "url_id". This is needed to implement static method *url_id_fixup*, which updates the values of changed **url_id**s, as discussed in Section 2.2.1.

Associated with each Table instance are the table's name and a Vector containing the columns.

Figure 4-4 shows the public methods associated with Table. They support the creation and deletion of SQL tables and provide access to the Column information.

4.3.3 Squeal internal tables

Not previously discussed are the Squeal internal tables, **creation** and **computation**. These are not visible to the Squeal user.

creation

The **creation** table, shown in Table 4.1, is used to keep track of user-defined tables across execution sessions. Table names are limited to 15 characters, while the definition string is unbounded. When the user creates a table, a line is added to **creation**, a new SQL table is created on the server, and a UserReadableTable instance is created. If a table is deleted, the corresponding line is removed from **creation**, the SQL table is dropped, and the instance is freed. Upon Squeal start-up, all of the entries in **creation** are interpreted, to re-create the UserReadableTable instances.

`public static Table createTable(DBstate dbs, String def)`
Create a Table instance from a definition string, creating a table on the SQL server if necessary
`public static void init(DBstate dbs)`
For each definition in the **creation** table, call **createTable**
`public static Table getTable(String n)`
Return the Table instance associated with a name, or `null`
`public static void dropTable(DBstate dbs, String name)`
Permanently delete a Table instance and the associated SQL table, given its name
`public boolean existsP(DBstate dbs)`
Check whether a table with name `name` exists on the SQL server
`public String createIfNecessary(DBstate dbs)`
Create on the SQL server a table with the characteristics of the instance variable, unless the table already exists
`public Column columnMatch(String arg)`
Return the Column instance associated with a name, or `null`
`public static void url_id_fixup(int old_url_id, int new_url_id, ...)`
In all columns of type **url_id**, replace values of `old_url_id` with `new_url_id`

Figure 4-4: Methods Defined for Table

computation		
colname	*type*	*notes*
compute_id	INT	primary key
stamp	DATETIME	
tab	CHAR(15)	
column	CHAR(15)	
value	VARCHAR(255)	
helper	CHAR(15)	
num	INT	

Table 4.2: The **computation** relation

computation

Purpose The **computation** relation is used to keeps track of what implicit or explicit FETCH statements have been performed and when. This can be used to prevent unnecessary recomputation of recently-computed information or to facilitate recomputation of stale information. Each line of the **computation** relation can be thought of as a thunk [1] bundled with the time of its execution and a pointer to its results.

Details The columns of the **compute** relation are shown in Table 4.2. Each **computation** relation has a unique numeric **compute_id**. Also stored are the table name, column name, and column value. For statements using **helper** and **num** arguments, these also appear in the table. The **page**, **rcontains**, **rlink**, and **tag** tables also have a **compute_id** column, not previously discussed. This provides information about when each line of these tables was created. The other tables do not have **compute_id** relations, because they can be deduced from tables on which they depend. While the information currently is not used, it would be easy to modify the system to reload pages that were judged to have expired.

compute_id	stamp	tab	column	value
7875	Oct 8 1997 6:19PM	**rlink**	**source_url_id**	&3&5&

Table 4.3: Entry in **computation** table created in processing the Squeal statement "FETCH rlink(**dest_url_id**=3&5)"

For a recomputation to be prevented, the old and potential **tab** and **column** columns must be identical, and the **value** columns must be identical except in the case of conjunctions and disjunctions. For conjunctions and disjunctions, the **value** string is the set of terms, delimited by "&" or "|", respectively. Table 4.3 shows the **computation** entry created by the Squeal statement:

FETCH rlink(**dest_url_id**=3&5);

If a subsequent delimited conjunction is a substring of the **value** in the table, e.g., "&3&5&", the computation is not redone. For disjunctions, the later **value** must be a superstring of the original **value**. In any case, the terms must appear in the same order for a match to be noted, since textual comparison is used.

The **helper** and **num** fields are set when **tab** is **rcontains** or **rvalue**, to indicate which Web search tool was used as a helper and how many items were requested. If the user issues two FETCH statements involving **rlink** that have different **helper** values, two Web requests will be made. If two statements are identical except for the **num** values, the second FETCH will only be performed if its **num** value is the greater one.

4.4 Exceptions

Class ParasiteException represents Squeal exceptions. It can be instantiated and has the following four subclasses:

- QuitException, thrown when the "quit" statement is executed.
- UnboundVariableException, thrown when a reference to an undefined variable is made.
- UnsupportedProtocolException, thrown when an attempt is made to retrieve a URL whose protocol is not "http".
- UnsupportedURLException, thrown when an attempt is made to retrieve a malformatted URL.

4.5 Representation of variables

4.5.1 SymbolTable

The SymbolTable class is built on top of java.util.HashTable. The hash table component is used to map variable names (of type String) to values (of type Object). A member variable **parent** of class SymbolTable is either set to null or to a parent SymbolTable. Methods are defined to get or put the value of a symbol as well as to check whether a symbol is defined (Figure 4-5).

4.5.2 Bindings

The Bindings class is a subclass of java.Util.Hashtable and is used to represent stack frames. Specifically, the inherited hashtable is used to represent formal-actual argument pairs and a parent symbol table. A Binding instance is created when a Squeal procedure call is made. It is also used to implement FETCH statements. Methods are defined to add, remove, access, or check for the existence of a binding, as shown in Figure 4-6. The methods that create **String**s are used to support the **Computation** table.

`public synchronized boolean containsKey(Object key)`
Return true if **super**.containsKey(**key**) is a key in the constituent hash table.
If not and if **parent** is null, return false. Otherwise, return **parent**.`containsKey`(**key**).
`public synchronized Object get(Object key)`
If **super**.containsKey(**key**), return **super**.get(**key**).
If not and if **parent** is null, return null. Otherwise, return **parent**.`get`(**key**).
`public synchronized Object put(Object key, Object val)`
Execute **super**.put(**key**, **val**).

Figure 4-5: Methods Defined for SymbolTable

`public synchronized Object put(Object key, Object val)`
Call `parent.put(key, val)`
`public synchronized Object get(Object key)`
If **super**.containsKey(**key**), return **super**.get(**key**).
If not and if **parent** is null, return null. Otherwise, return **parent**.`get`(**key**).
`public synchronized Object remove(Object key)`
Call `parent.remove(key)`
`public void extend(Vector v)`
For each namedArg **na** (Section 4.9.1) in **v**,
call `put(na.name, na.value)`.
`public synchronized boolean containsArg(Object key)`
Return true if **super**.containsKey(**key**) is a key in the constituent hash table.
If not and if **parent** is null, return false. Otherwise, return **parent**.`containsKey`(**key**).
`public String toConjunction()`
Return a string representing each key-value pair (k,v) as "$< k >=< v >$",
delimited by " AND " (e.g., "x = 1 AND y = 2").
`public String keysString()`
Return a comma-separated list of the keys in the constituent hash table.
(This if for debugging purposes.)
`public String valuesString()`
Return a comma-separated list of the values in the constituent hash table.
(This if for debugging purposes.)

Figure 4-6: Methods Defined for Bindings

- SimpleNode
  - NodeWithRequiredName
  - NodeWithOptionalName
  - NodeContainingList
    - NodeContainingParenthesizedList
  - BinaryNode

Figure 4-7: Class Hierarchy of Parser-Generated Nodes

4.6 Parser

Squeal is parsed by the Java Compiler Compiler (JavaCC) [36]. The Squeal grammar, with actions, is shown in Appendix B. JavaCC creates LALR parsers with lookahead one, except when greater lookaheads are explicitly specified in a region of the grammar. The maximum lookahead needed for the Squeal grammar is two.

This section describes the data objects produced by the parser, all of which are subclasses of SimpleNode, as shown in Figure 4-7. The parser prepends "AST" (for "abstract syntax tree") to terminal names to create class names. Table 4.4 shows SimpleNode and its direct subclasses. Parser node classes are shown in Table 4.4.

4.6.1 SimpleNode

The class SimpleNode is provided by JavaCC and modified for Squeal. Methods provided by JavaCC provide access to child nodes. A key method defined for Squeal is *toSQL*, which converts a SimpleNode to a SQL representation so it can be passed to the SQL server. If the node has only one child, the return value is the result of calling *toSQL* on the child. If there are multiple children, the results of recursive calls to *toSQL* are concatenated, separated by space characters. Subclasses of SimpleNode either inherit *toSQL* (as in the case of ASTnumericLiteral, which recursively calls *toSQL* for its one child) or override it (as in the case of ASTcell, which prints its two children separated by a period). Other methods allow a child node to be removed (in order to get rid of an unneeded argument to FETCH) and to find all the tables, cells, or variables referred to in a statement, which is necessary when interpreting SELECT statements. A complete list of methods is shown in Figure 4-8. The following sections describe methods defined for subclasses of SimpleNode.

4.6.2 NodeWithRequiredName

Because so many productions contain a string that needs to be retained in addition to the node type, there is a class NodeWithRequiredName, of which ASTfuncall is a subclass, with the name field set to the function name. NodeWithRequiredName defines methods *setName* and *getName* and overrides methods *toString* and *toSQL* to include the name.

4.6.3 NodeWithOptionalName

NodeWithOptionalName is similar to NodeWithRequiredName, except that setting the name field is optional. The methods *setName* and *getName* are defined, and *toString* is overridden, returning the name if one exists and the empty string otherwise.

4.6.4 NodeContainingList

NodeContainingList is used to represent nodes that consist of a list of child nodes. Specifically, it represents ASTselectList and ASTtableList. It overrides methods *toString* and *toSQL* and defines

```
public String toString()
```
Return the identifier associated with the instance (JavaCC)
```
public int jjtGetNumChildren()
```
Return the number of children of the node (JavaCC)
```
public SimpleNode jjtGetChild(int i)
```
Return the i^{th} child of the node (JavaCC)
```
public Object toSQL(SymbolTable symtab)
```
Return a legal SQL representation of the node and its children
```
public void findTableNames(Vector v)
```
Build a vector containing the tables referenced by this node and its children
```
public void findCells(Vector v)
```
Build a vector containing the ASTcells referenced by this node and its children
```
public void findVariables(Vector v)
```
Build a vector containing the ASTvariables referenced by this node and its children
```
public void removeChild(SimpleNode child)
```
Remove the specified child node

Figure 4-8: Methods Defined for SimpleNode, either by JavaCC (as indicated) or purely for Squeal.

SimpleNode	NodeWithRequiredName	NodeWithOptionalName
ASTcell	ASTcolumnDef	ASTaggregateExpression
ASTcolumnsList	ASTcomputeStatement	ASTselectItem
ASTconvertExpression	ASTcreateStatement	
ASTinputStatement	ASTdeffuncStatement	**NodeContainingList**
ASTlistExpression	ASTdefprocStatement	ASTselectList
ASTnamedArgument	ASTdeleteStatement	ASTtableList
ASTnamedArgumentList	ASTdescribeStatement	
ASTnegationExpression	ASTdropStatement	**NodeContainingParenthesizedList**
ASTnumericLiteral	ASTfuncall	ASTargList
ASTorderItem	ASTgroupbyDef	ASTsymbolList
ASTorderList	ASThavingDef	
ASToutputStatment	ASThelpStatement	**BinaryOperation**
ASTparenthesizedExpression	ASTinsertStatement	ASTconjunctionExpression
ASTprintStatement	ASTletStatement	ASTdisjunctionExpression
ASTquitStatement	ASTorderbyDef	ASTlogicExpression
ASTrelExpression	ASTrelRHS	ASTproductExpression
ASTselectStatement	ASTstringLiteral	ASTsumExpression
ASTstar	ASTsymbolLiteral	
ASTstatement	ASTtableName	
ASTunaryExpression	ASTvariable	
ASTupdateStatement	ASTwhereDef	

Table 4.4: Classes of Nodes Created by Parser

method *toVector*, which creates a Vector, each of whose elements is the SQL representation of a child node.

4.6.5 NodeContainingParenthesizedList

NodeContainingParenthesizedList is a subclass of NodeContainingList. It overrides *toSQL* to include parentheses around the list. It is used to represent ASTargList and ASTsymbolList.

4.6.6 BinaryOperation

BinaryOperation is used to represent binary logical and arithmetic operations. The *setOp* method is used to set the operator, and *toSQL* is overridden to call *doBinaryOp*, which, in the case of arithmetic operations, performs the operation if the values of both operands are known, otherwise returning the operator information with the operand in between, to eventually be passed to the SQL server.

4.6.7 Context

The behavior of *toSQL* may depend on the context in which it is called. For example, if the variable "x" is undefined within Squeal, the result of calling *toSQL* on the ASTvariable representing "x" depends on how it is used:

- If the entire statement is "PRINT **x**", then an UnboundVariableException should be thrown.
- If the entire statement is "SELECT **x** FROM usertable", the statement should be passed to the SQL server.

In client mode, the UnboundVariableException would be thrown; in server mode, the representation appropriate for server access is used. Another use is to provide proper syntax for **String**s. For example, the interpretation of **String** s, bound to "foo" depends on whether the value will be sent to the SQL server:

- If the entire statement is "userfunc(s)", *s* should be interpreted as **foo** (without quotes).
- If the entire statement is "SELECT * FROM usertable WHERE col = s", then *s* should be interpreted as **'foo'** (with single quotes).

The Context class has a single instance, which contains a stack, the topmost element of which indicates whether the system is in "client" or "server" mode. The appropriate value is pushed on to the stack when a statement is interpreted. By default, it is in "client" mode. The value "server" is pushed onto the stack upon interpretation of an INSERT, DELETE or UPDATE statement, and popped at the end of its interpretation. Similarly, "client" is pushed at a function call (in case one is nested in an INSERT statement, for example).

4.7 Output

Squeal supports five output streams, shown in Table 4.5. They are built on top of a hierarchy of subclasses of java.io.PrintWriter, shown in Figure 4-9. NullPrintWriter overrides all of PrintWriter's **write**, **print**, and **println** methods to ignore their arguments and do nothing. It is used for an output stream that the user is not interested in viewing. The class LoggingPrintWriter is instantiated for streams that the user wishes to view. The constructor takes three arguments: an output stream (e.g., **System.out**), a name (e.g., "status"), and a PrintWriter **logStream**, to which it echoes whatever it prints, with the stream name prepended. By default, **logStream** is of class NullPrintWriter and no log is created. If the user requests a log via the command-line interface (Section 3.6), it is of type LogPrintWriter, which prints everything it is passed to a file, with a time stamp attached. Figure 4-10 shows an excerpt from a sample log file. The numeric column is the number of milliseconds since the start time, divided by 128.

name	default value	purpose
StatusStream	System.out	prompts and other user-interface communication
OutputStream	System.out	results of computations
DebugStream	(none)	messages for debugging purposes
ErrorStream	System.err	errors
LogStream	(none)	all of the above

Table 4.5: Streams Used by Squeal

java.io.PrintWriter
- NullPrintWriter
- LoggingPrintWriter
- LogPrintWriter

Figure 4-9: Class Hierarchy Based on java.io.PrintWriter

0		Creating log at Wed Jul 09 17:16:36 PDT 1997
10	status	Opening database
30	debug	Push: AccessibleBufferedInputStream1dc613f5
30	debug	Debug: Completed initialization
30	status	->
33	debug	Calling processTree with type class squeal.ASTstatement
33	debug	Calling processTree with type class squeal.ASTdeffuncStatement

Figure 4-10: Sample Log Output

```
public static void main(String[] args)
```
Top-level procedure, containing the call to *init* and the read-eval-print loop.
```
public static void init(String[] args) throws ex.ParasiteException
```
Code to process the command-line arguments, set up the input streams, and call other initialization routines.
```
public static void printUsage()
```
Print information about the command-line interface (section 3.6).
```
public static Object processTree(SimpleNode node, SymbolTable symtab)
        throws ex.ParasiteException
```
Return the result of evaluating the parsed Squeal statement or expression in *node*.
```
public static Object inputFrom(Object argument) throws ex.ParasiteException
```
Create an input stream to the specified file.
```
static boolean popInputStack() throws java.io.IOException
```
Pop the stack of input streams, indicating that input from a stream is complete. Return *true* if more input streams are on the stack, *false* if it is empty.
```
public static Object Interpret(String s)
```
Called in order to cause a String to be interpreted as a Squeal statement. *Interpret* calls the parser and then *processTree*.
```
public static Object get(String s)
```
Look up symbol *s* in the top-level symbol table, *globSymtab*.

Figure 4-11: Methods in FrontEnd

4.8 FrontEnd

The class FrontEnd contains the code to start the Squeal interpreter and to repeat the read-eval-print loop. During initialization, a member variable *globalSymtab* of class SymbolTable is created, which holds environment variables (section 3.6 and any values defined at the top-level in the interpreter. Another member variable, *inputStack*, maintains a stack of input streams, initially holding only standard input. Almost all of program execution occurs within the read-eval-print loop. In each iteration, a statement is read from the topmost input stream. If it is a file inclusion statement (*in*), a handle to the specified file is pushed onto the stack. Otherwise, the statement is passed to *processTree*, which consists of a giant case statement to handle different types of Squeal statements. If *processTree* throws a QuitException, then the input stack is popped and execution continues with the new top input stream (or execution ends if the stack is now empty). If *processTree* returns a value, it is printed, and the loop is repeated. FrontEnd also contains the method *Interpret*, which passes a String to the parser and executes it. The *get* method is used when other classes need to access *globalsymtab*. Figure 4-11 shows a complete list of methods in FrontEnd.

4.9 Miscellaneous

4.9.1 namedArg

The class namedArg is used by parsing routines to represent a formal-actual parameter pair, such as for the statement: "FETCH(**url_id**=7)". The member variables **name** and **value** contain the formal and actual parameters, respectively, "**url_id**" and 7 in this example. **namedArg** instances are later used for creating Bindings instances.

`public Vector colNames`
  The names of the columns
`public Vector colsizes`
  The maximum width of each column
`public Vector rows`
  The set of rows, each element of which is a java.util.Vector containing the elements of each row `public int numCols`
  The number of columns

Figure 4-12: Member Variables for SelectionResult

4.9.2 Cell

The class Cell is used to represent a cell in a SQL table, e.g., ("utable.colfoo"). Member variables **alias** and **column** contain the table alias and column, respectively, "utable" and "colfoo" in this example.

4.9.3 Junction

The abstract class Junction is used as a superclass for Conjunction and Disjunctions, which are used to represent conjunctions and disjunctions in FETCH statements. Each instance of Junction contains two member variables: **terms**, a java.util.Vector whose elements are the conjuncts/disjuncts, and **separator**, which is either "&" or "|". The method *toString* converts a Junction instance to its printed representation, e.g., "**url_id** = 6 | **url_id** = 7", and is used in constructing queries for the SQL server. The method *toQuotedString* is similar but quotes each term. The method *toList* returns a string containing a comma-separated list of the terms, useful for construction IN clauses of SQL statements.

4.9.4 Set

The class Set provides a simple implementation of mathematical sets. It implements methods *addElement*, *union*, *size*, and *contains*, as well as *elements*, which returns an Enumeration of the elements.

4.9.5 SelectionResult

The class SelectionResult is used to represent the results of a SQL SELECT query. Its member variables are shown in Figure 4-12. The method *display* prints the result to the specified java.io.PrintWriter. The method *toObject* returns the element of the table, if there is only one, and throws a ParasiteException otherwise. It is used for statements of the form: "LET n = (SELECT MAX(**url_id**) FROM **link**".

4.10 Functions and Procedures

4.10.1 UserCallableFunc

Figure 4-13 shows the class hierarchy for user-callable functions, based at UserCallableFunc, which is an abstract type. UserCallableFunc has a pair of static hash tables, **funcHash** and **helpHash**, used to map function names to the function object and help string, respectively. Methods, including constructors (which we have not usually shown), can be found in Figure 4-14.

UserCallableFunc
    UserDefinedFunc
    UserDefinedProc
    JavaDefinedFunc

Figure 4-13: Class Hierarchy for Functions and Procedures

`public UserCallableFunc(String fName, String helpString)`
    Add **(fName, this)** to the function look-up table (**funcHash**) and
    add **(fName, helpString)** to the help look-up table (**helpHash**).
`public UserCallableFunc(String fName)`
    Add **(fName, this)** to the function look-up table (**funcHash**) and
    add **(fName, "no help available")** to the help look-up table (**helpHash**).
`abstract public Object call(SymbolTable s, Vector v, DBstate dbs)`
    Apply the associated function with symbol table **s**, actual parameters and
    database statement **dbs**.
`public static UserCallableFunc getFunc(String fName)`
    Return the function object associated with **fName** (or null, if none is defined
`public static Object call(String s, SymbolTable symtab, Vector v, DBstate dbs)`
    Apply the function named **s** to symbol table **symtab**, actual parameters **v**,
    and database state **dbs**, throwing a Parasite Exception if no such function exists

Figure 4-14: Methods Defined for UserCallableFunc

```
public class JDFstrcat extends JavaDefinedFunc {
    public JDFstrcat() {
        super("strcat", "Concatenate any number of strings");
    }

    public Object call(Vector v, DBstate dbs) {
        StringBuffer outputSB = new StringBuffer();

        for (int i = 0; i ¡ v.size(); i++) {
            Object curObj = v.elementAt(i);
            outputSB.append(Utils.unquote(curObj.toString()));
        }
        return new String(outputSB);
    }
}
```

Figure 4-15: Java-Defined Function Strcat

4.10.2 UserDefinedFunc

The class UserDefinedFunc is used to represent functions defined by the Squeal user. As discussed in Section 3.3, the body of a function is a single expression. When the function is created, the argument list and body of the function are stored in hashed tables, keyed by the new function's name. Upon application, the system verifies that the number of actual parameters is equal to the number of formal parameters and extends the symbol table to include these bindings, then executing the procedure body and returning the result.

4.10.3 UserDefinedProc

The class UserDefinedProc is used to represent procedures defined by the Squeal user. As discussed in Section 3.3, the body of a procedure is a sequence of statements. The behavior of UserDefinedProc is identical to UserDefinedFunc, except that it is the value of the last *statement* (as opposed to the sole *expression*) that is returned.

4.10.4 JavaDefinedFunc

The class JavaDefinedFunc is used as a superclass for user-callable functions defined in Java. It contains the following abstract function, which must be defined in all its subclasses:

```
abstract public Object call(Vector v, DBstate dbs)
```

Figure 4-15 shows the Java code defining JDFstrcat, a function to concatenate strings. Table 3.1 on page 36 shows a list of all current subclasses of JavaDefinedFunc.

4.10.5 SQLfunc

The abstract class SQLfunc is used to represent relations that can be called explicitly through *fetch* or implicitly through *select*. For example, the class SQLfuncLink represents the **link** relation. Note that SQLfunc is not a subclass of UserCallableFunction and that its subclasses cannot be called like regular functions. The key abstract function that must be overridden in any subclass is:

```
abstract public Boolean call(Bindings bindings, DBstate dbs, int compute_id)
```

```
public class SQLfuncRlink extends SQLfunc
{
  public SQLfuncRlink(DBstate dbs) {
    super("rlink", dbs);
    createAutomaticTable(dbs, "rlink",
        "source_url_id url_id, *anchor varchar(255), *dest_url_id url_id");
  }

  public Boolean call(Bindings bindings,
                      DBstate dbs,
                      int compute_id) throws ex.ParasiteException {

    String helper = (String)bindings.get("helper");
    SearchEngine se = SearchEngine.getFunc(helper);
    int num = ((Integer)bindings.get("num")).intValue();
    if (bindings.containsKey("anchor"))
      return se.computePagesContainingAnchor(dbs, compute_id,
                                             num, bindings.get("anchor"));
    else if (bindings.containsKey("dest_url_id"))
      return se.computePagesReferencingDest(dbs, compute_id,
                                            num, bindings.get("dest_url_id"));
    throw new ex.ParasiteException("Unable to process this call to rlink");
  }
}
```

Figure 4-16: Code for SQLfuncRlink

The procedure takes an item of class Bindings (Section 4.5.2) representing the known assignments, the database state, and a **compute_id** (Section 4.3.3). It then adds rows as appropriate to the eponymous relation or throws an exception if unable to do so.

Figure 4-16 shows the implementation of SQLfuncRlink. In the constructor *SQLfuncRlink*, the automatic table (Section 3.5.2) **rlink** is created. Asterisks indicate defining columns 3.4 **anchor** and **dest_url_id**. The procedure call determines the **helper** and **num** bindings and then checks whether **anchor** or **dest_url_id** is bound, calling the corresponding function. If neither is bound, a ParasiteException is thrown.

4.11 SearchEngine

The abstract class SearchEngine encapsulates search engines that can be called by the system. The static method *getFunc*, used in Figure 4-16 maps from a string representing a search engine to the object. Instance methods are shown in Figure 4-17.

4.11.1 AltaVista

The AltaVista class supports all of the SearchEngine methods. For *computePagesContainingText* and *computePagesContainingAnchor*, **value** can be of type String or Junction. For *computePages-ReferencingDest*, **value** can be an Integer, representing a **url_id**, or a Junction. Any other types for **value** yield an exception.

```
abstract public Boolean computePagesContainingText(DBstate dbs, int compute_id,
        int num, Object value)
```
Add to **rcontents** up to **num** pages reported to contain **value**, returning true on success, false otherwise.

```
abstract public Boolean computePagesContainingAnchor(DBstate dbs, int compute_id,
        int num, Object value)
```
Add to **rlink** up to **num** pages reported to contain a hyperlink with anchor text **value**, returning true on success, false otherwise.

```
abstract public Boolean computePagesReferencingDest(DBstate dbs, int compute_id,
        int num, Object value)
```
Add to **rcontents** up to **num** pages reported to contain a hyperlink with destination **value**, returning true on success, false otherwise.

Figure 4-17: Methods Defined for SearchEngine

4.11.2 Lycos

The Lycos class supports *computePagesContainingText* with **value** of type String or Junction. Calls to the other functions yield an exception, because Lycos does not support those types of searches.

4.12 Computation

The class Computation supports the FETCH statement, whether it is entered by the user or produced in interpreting a SELECT statement. For example, consider the following FETCH statement:

```
FETCH link(url_id=possible_id) FROM utable u WHERE u.score > 6;
```

These are the steps that take place in the public function *performComputation*:

1. If multiple `namedArgument` items are given, throw out all but the one whose **name** appear earliest in the table definition, retaining **helper** and **num** assignments.
2. Use SimpleNode.*findCells* (Section 4.6.1) to determine what columns are needed from the SQL server (e.g., **utable.possible_id**).
3. Request these columns (e.g., "SELECT u.possible_id FROM utable u WHERE u.score>6"), putting the results in SelectionResult **sr**.
4. For each row of **sr**
   (a) Bind each cell name (e.g., **u.possible_id**) to the row's corresponding value (e.g., 6).
   (b) If a row does not yet exist in COMPUTATION representing this computation:
      i. Call the appropriate function (**link**) with the appropriate bindings (**url_id**=6).
      ii. Add the appropriate line to COMPUTATION

4.13 Selection

The Selection class contains the methods and data for interpreting SELECT statements according to the algorithms described in Section 3.5.2. Each Selection instance consists of a Table, column name, relational operator (i.e., "=" or "LIKE"), and value, plus a Vector of aliases of Tables that must be bounded before the Selection can be determined. Consider the interpretation of the following statement from the Home Page Finder:

```
SELECT DISTINCT u.url_id, 20, 'Base of file name same as name of directory: '
     + v.vcvalue           FROM urls u, parse p1, parse p2, valstring v
WHERE u.url_id IN (SELECT DISTINCT url_id FROM candidate)
AND p1.url_value_id = u.value_id     AND p2.url_value_id = u.value_id
AND v.value_id = u.value_id      AND p1.depth+1 = p2.depth
AND (p1.value = p2.value + '.html'   OR p1.value = p2.value + '.htm');
```

Execution of *performSelectStatement* is as follows:

1. Local variable *unboundedTables* is set to the table aliases (e.g., {u, p1, p2, v})
2. If all tables are user-readable (section 3.5.1), send the statement to the SQL server and return. (In this instance, (*u* (**urls**) and *v* (**valstring**) are user-readable; *p1* and *p2* (**parse**) are not.)
3. If no WHERE clause is present, throw a ParasiteException with the message "In automatic mode, select statements of canonical tables must contain a nontrivial where clause."
4. Group together any conjunctions referring to the same column of the same alias. (In this instance, there are none.)
5. Local variable *numRemaining* is set to *unboundedTables.size*() (e.g., 4)
6. While *numRemaining* > 0
   (a) Call *buildComputeClauses*, which attempts to provide a bound on a member of *unboundedTables*. Procedure *buildComputeClauses* loops through passed variable *unboundedTables* until it finds a table that is bound (i.e., has a defining column set to a known value), as shown by the pseudo-code in figure 3-13. As a side-effect, it removes the bound element from *unboundedTables* and creates **Selection** instances, placing them in *selectionTable*.
   (b) If no tables were bound (i.e., *unboundedTables.size*() = *numRemaining*, throw a ParasiteException with the message: "Unable to provide bound on canonical tables: " + *unboundedTables*; otherwise, decrement *numRemaining*

   In our example, the tables are bounded in this order:

   (a) u, creating the following Selections:
      i. {name = "u", table = **urls**, lvalue = "u.**url_id**", op = "LIKE", rvalue = "(SELECT DISTINCT url_id FROM candidate candidate0)", dependences = { } }
      ii. {name = "u", table = **urls**, lvalue = "u.**value_id**", op = "=", rvalue = "v.value_id", dependences = {v} }
   (b) p1, creating the Selection {name = "p1", table = **parse**", lvalue = "p1.**url_value_id**", op = "=", rvalue = "u.**value_id**", dependences = {u} }
   (c) p2, creating the Selection {name = "p2", table = **parse**", lvalue = "p2.**url_value_id**", op = "=", rvalue = "u.**value_id**", dependences = {u} }
   (d) v, creating the Selection {name = "v", table = **valstring**", lvalue = "v.**value_id**", op = "=", rvalue = "u.**value_id**", dependences = {u} }
7. Determine the order in which the FETCH statement corresponding to each Selection must be computed, honoring the *dependences* values. In our example, an acceptable order is {u, p1, p2, v} because *p1*, *p2*, and *v* depend only on *u*.
8. For each Selection in *selectionTable* that is not a user-readable table, in order:
   (a) Construct the FETCH statement corresponding to the Selection, appending "where (1=1)".
   (b) Add constraints inherited from the dependences (shown in italics below).
   (c) Interpret the FETCH statement. In our example, these would be:

i. "FETCH parse(url_value_id = u.value_id) FROM urls u, valstring v WHERE (1=1) *AND u.url_id IN (SELECT DISTINCT url_id FROM candidate candidate0) AND u.value_id = v.value_id AND v.value_id = u.value_id*"

ii. "FETCH parse(url_value_id = u.value_id) FROM urls u, valstring v WHERE (1=1) *AND u.url_id IN (SELECT DISTINCT url_id FROM candidate candidate0) AND u.value_id = v.value_id AND v.value_id = u.value_id*"

9. Pass the original SELECT statement to the SQL server, which is now able to answer the query.

4.14 Utils

The class Utils contains many useful utility functions, for string processing, node manipulation, Web access, and SQL server access. It also provides an *assert* method, which, if its first argument is false, prints an error message and halts Squeal.

4.14.1 String Manipulation

A number of routines, shown in Figure 4-18, support general string manipulation and conversion among different formats, e.g., Squeal, SQL, and URL formats. For example, percentage signs are used in SQL queries to represent wild cards; *unpercent* removes them for Web queries. *HTMLify* converts Strings into a format appropriate for appearing in a URL, such as by replacing every space character (" ") with a plus sign ("+"). Similarly, to prepare a statement for the SQL server, special characters are protected by *doubleQuotes*, which replaces each single quotation mark with a pair of single quotation marks, and *quotePercents*, which puts brackets around each percentage sign The method *vectorToCommaSeparatedString*. The method *cleanText* assures that Strings returned from the SQL server are properly bounded.

4.14.2 SQL Server Access

Utils provides useful methods for accessing the SQL server, shown in Figure 4-19. The method *getSoleSelection* is used to make a query that returns a single result, such as the **vcvalue** associated with a defined **value_id**. If zero or more than one results are returned, a ParasiteException is thrown. (How this is handled is discussed in the next section.) The method *getSoleSelectionNoFail* is similar but calls *assert* to ensure that one and only one result is returned. It is used for system queries that should always return exactly one result, such as:

```
SELECT MAX(value_id) FROM valstring
```

The method *tableExists* checks whether a table exists on the SQL server. The five versions the method *doInsert* are used for inserting data in a variety of formats into a SQL table. The method *executeUpdate* is used to send an already-constructed String to the SQL server.

4.14.3 Conversion

Utils contains many routines (Figure 4-20) to convert from one type of data to another; for example, from the String "foo" to the associated **value_id**. Most conversions are implemented as simple queries, which are passed to *getSoleSelection*, such as:

```
SELECT value_id FROM valstring WHERE vcvalue = "foo"
```

If the operation fails, the appropriate modification is made to the proper tables (e.g., adding a line to **valstring**) and the query is re-attempted.

```
public static String stringSubstituteFirst(String origString, String origSub,
        String newSub)
```
Replace the first instance of *origSub* with *newSub* in *origString*.
```
public static String stringSubstitute(String origString, String origSub,
        String newSub, boolean globalP)
```
Replace every (if *globalP* is true) or the first (if *globalP* is false) occurrence of *origSub* with *newSub* in *origString*.
```
public static String makeBound(String s, int limit)
```
Return a String containing the largest left substring of *s* such that its length is less than *limit*.
```
public static String unquote(String s)
```
Remove quotation marks from the beginning and end of *s* if any are present.
```
public static String unpercent(String s)
```
Remove percentage signs from the beginning and end of *s* if any are present.
```
public static String HTMLify(String s)
```
Convert a String into a format appropriate for using as a URL, e.g., replacing "," with "%2C", etc.
```
public static String doubleQuotes(String s)
```
Make a legal SQL statement by replacing every single quote with a pair of single quotes.
```
public static String quotePercents(String value)
```
Replace percentage signs (%) in *value* with [%] in preparation for passing a statement to the SQL server. This effectively quotes the percentage sign.
```
public static String cleanText(String s)
```
Remove any characters appearing after the null character in *s*.

Figure 4-18: String-Manipulation Methods Defined for Utils

```
public static Object getSoleSelection (String sqlString, DBstate dbs)
        throws ex.ParasiteException
```
Pass the SQL statement *sqlString* to the SQL server and return the result. If more than one result is returned by the SQL server, throw a ParasiteException.
```
public static Object getSoleSelectionNoFail (String sqlString, DBstate dbs)
```
Like *getSoleSelection*, but assert failure (halting the system) if more than one value is returned.
```
public static boolean tableExists(String tableName, DBstate dbs)
```
Check whether a table named *tableName* exists on the SQL server.
```
public static void doInsert(...)
```
Create an INSERT statement assigning the specified columns to the specified values for the indicated table and send the statement to the SQL server.
```
public static void executeUpdate(DBstate dbs, String q) throws ex.ParasiteException
```
Send the String *q* to the SQL server.

Figure 4-19: SQL Server Access Methods Defined for Utils

```
public static int String_to_value_id(DBstate dbs, String value) throws
        ex.ParasiteException
```
Return the **value_id** of the line of **valstring** with *value* as the associated text, creating the line if necessary.

```
public static String value_id_to_String(DBstate dbs, Integer value_id)
        throws ex.ParasiteException
```
Return the text of the line of **valstring** that has *value_id* in the **value_id** field.

```
public static int urlstring_to_url_id(DBstate dbs, String urlstring)
        throws ex.ParasiteException
```
Return the **url_id** of the line of **urls** corresponding to the String *urlstring*.

```
public static int URL_to_url_id(DBstate dbs, URL url, ...)  throws
        ex.ParasiteException
```
Return the **url_id** corresponding to *url*.

```
public static int URL_to_value_id(DBstate dbs, URL url, int compute_id)
        throws ex.ParasiteException
```
Return the **value_id** associated with the string representation of *url*.

```
public static int value_id_to_url_id(DBstate dbs, int value_id, ...)
        throws ex.ParasiteException
```
Return the **url_id** corresponding to the URL whose string representation has a **value_id** of *value_id*.

```
public static URL url_id_to_URL(DBstate dbs, int url_id, ...)  throws
        ex.ParasiteException
```
Return the URL whose **url_id** is *url_id*.

```
public static String url_id_to_urlstring(DBstate dbs, int url_id, ...)
        throws ex.ParasiteException
```
Return the string representation of the URL whose **url_id** is *url_id*.

```
public static String URLtoContentString(URL url, DBstate dbs) throws
        ex.ParasiteException
```
Return the String corresponding to the contents of the page addressed by *url*.

Figure 4-20: Conversion Methods Defined for Utils

```
public static Vector ProcessChildrenVector(SimpleNode node, SymbolTable symtab)
```
Return a Vector containing the results of interpreting every child of *node*.
```
public static Vector ProcessChildrenVectorSQL(SimpleNode node, SymbolTable symtab)
```
Return a Vector containing the results of executing *toSQL* on every child of *node*.
```
public static SimpleNode scoot(SimpleNode sn)
```
Return the earliest descendant of SimpleNode *sn* that has either zero or multiple nodes (i.e., does not have only one child).

Figure 4-21: Methods Defined for Utils

4.14.4 Node Manipulation

Util contains a set of routines for operating on the SimpleNodes returned by the parser. The method *ProcessChildrenVector* calls FrontEnd.*processTree* on all of the children of the passed node, building up a Vector. This is used in interpreting FETCH statements and making Squeal function/procedure calls. The method *ProcessChildrenVectorSQL* is similar, except it evaluates nodes with SimpleNode.*toSQL*. It is used for parsing INSERT statements. The method *scoot* walks down the descendent tree from a node until it reaches a node with zero or more than one children, which it returns. It is used to quickly traverse single strands in the parse tree. These methods are listed in Figure 4-21.