Chapter 3

Squeal

The previous chapter described a relational database model of the Web. This abstraction is implemented by the Squeal interpreter, which determines what information from the Web needs to be brought into the database and how to do so. This chapter describes the Squeal language and the algorithms that support its interpretation. The next chapter goes into lower-level detail about the Java implementation of the Squeal interpreter.

The Squeal core is syntactically a subset of SQL, with a different semantics. For example, consider the possible interpretations of the following statement:

```
SELECT dest_url_id FROM link WHERE source_url_id = 7
```

The semantics of the statement in SQL are to query the **link** table and return to the user all values of **dest_url_id** whose associated **source_url_id** is seven. No changes are made to the database by the statement's execution. In contrast, the semantics in Squeal are to return the **url_id**s of all links from the Web page represented by **source_url_id**. As a side effect, the **link** table in the database representation may be modified, but users need not be aware of this. The purpose of Squeal is to give users the illusion that they are directly querying the Web.

When a statement is entered, the Squeal interpreter parses it, rejecting it with an error message if it is invalid and generating a parse tree otherwise. If the statement doesn't reference any of the special tables described in Chapter 2, it is passed through to the SQL server and the reply is passed back to the user. If special tables are referenced, the Squeal interpreter determines what information from the Web is needed in order to answer the query. If the information is not yet in the database, it fetches and parses Web pages and puts the needed information into the database. The rest of this chapter gives the syntax and semantics of Squeal and the algorithms for its interpretation. In addition to the core, which only includes legal SQL statements, Squeal contains some simple mechanisms for naming and abstraction, also described in this chapter.

3.1 Lexical Tokens

Figure 3-1 shows the lexical tokens used in the Squeal grammar. Strings may be delimited with either single or double quotation marks. Identifiers may start with optional "#" characters (for SQL temporary tables), then must have a letter, and then may be followed by any combination of letters, digits, and underscores. The language is case-insensitive. Statements are separated by semicolons, not shown in the below grammars. Comments are begun with a double slash ("//") and extend to the end of a line.

3.2 Expressions

The grammar for expressions is shown in Figure 3-2. The semantics of expression evaluation depend on whether an object is a first-class or passed-through data type. The complete Squeal grammar

```
<STRING>:     ("'" (~["'"])* "'") | ("\"" (~["\""])* "\"")

<LETTER>:     ["A"-"Z", "a"-"z"]

<DIGIT>:      ["0"-"9"]

<IDENTIFIER>: ("#")* (<LETTER>)+ (<LETTER>|<DIGIT>|"_")*

<NUMBER>:     (<DIGIT>)+
```

Figure 3-1: Lexical tokens

appears in appendix B. In this chapter, we simplify it slightly for expository purposes.

3.2.1 First-class data types

Integers and strings are first-class objects. The binary operations "+", "-", "*", and "/" are defined for integers, as is unary minus. Strings may be bounded with single or double quotation marks.

3.2.2 Passed-through data types

Squeal also recognizes table types, aliases, column names, relational operators, and logical expressions but does not evaluate them, instead passing them to the SQL server for interpretation, typically as part of the WHERE clause of a SELECT statement.

3.2.3 Special data types

While the representation of the **url_id** and **value_id** types are integers, they are treated as special for two reasons:

1. It is convenient for the interpreter to know they are not ordinary integers, so the corresponding string can be displayed to the user.
2. The **url_id** associated with a specific URL (specifically, associated with a specific **value_id**) can change, as discussed in Section 2.2.1.

If the user defines a column to be of type **url_id**, the interpreter knows to change the values if the **url_id** changes.

3.3 Simple Statements

Figure 3-3 shows the grammar for SQL statements. In this section, we will discuss the statements that do not access the Web or the database.

3.3.1 Basic statements

The grammar for the **print** and **let** statements is shown in Figure 3-4. The **print** statement, which can be abbreviated "?", prints an expression. For example, "print 2*3" prints 6. The **let** statement sets a variable to an expression value. If the variable has not been defined, it is created. Any variable can hold either a string or an integer. The expression in the optional "ELSE" clause is evaluated if the first expression is null. This is similar to the "COALESCE" statement in SQL.

```
expression:                 disjunctionExpression

disjunctionExpression:      conjunctionExpression ("OR" disjunctionExpression)?

conjunctionExpression:      negationExpression ("AND" conjunctionExpression)?

negationExpression:         ("NOT")? relExpression

relExpression:              sumExpression (rel_opsumExpression)?

rel_op:                     "<" | "=" | ">" | "<>" | ">=" | "<="
                            | "NOT IN" | "LIKE" | "NOT LIKE" | "IN"

sumExpression:              productExpression (("+")|("-") sumExpression)?

productExpression:          unaryExpression (("*")|("/") productExpression)?

unaryExpression:            ("-")? parenthesizedExpression

parenthesizedExpression:    "(" selectStatement ")"
                            | "(" expression ")"
                            | funcall
                            | cell
                            | <NUMBER> | <IDENTIFIER> | <STRING>
                            | aggregateExpression
                            | "*"

cell:                       <IDENTIFIER> "." (<IDENTIFIER> | "*")

aggregateExpression:        aggregate_op "(" agg_restrict? expression ")"

aggregate_op:               "AVG" | "MAX" | "MIN" | "SUM" | "COUNT"

agg_restrict:               "ALL" | "DISTINCT" | "UNIQUE"
```

Figure 3-2: Grammar for Expressions. . The symbol "|" represents disjunction. When an item is followed by "?", it is optional; when followed by "*", it may appear any number of times, including zero. The nonterminal `funcall` is defined below in Figure 3-5.

```
statement: printStatement
         | letStatement
         | callStatement
         | helpStatement
         | deffuncStatement
         | defprocStatement
         | inputStatement
         | outputStatement
         | quitStatement
         | fetchStatement
         | selectStatement
         | createStatement
         | dropStatement
         | deleteStatement
         | updateStatement
         | insertStatement
         | describeStatement
```

Figure 3-3: Grammar for **statement**

`letStatement` = "LET" <IDENTIFIER> "=" `expression` ("ELSE" `expression`)

Figure 3-4: Grammar for LET Statements. The variable is defined to be the value of the first expression, unless it is null and an ELSE clause is present, in which case the second expression is used.

```
callStatement:    funcall

funcall:          <IDENTIFIER> argList

argList:          "(" (expression ("," expression)*)? ")"

deffuncStatement: "DEFFUNC" <IDENTIFIER> symbolList expression

defprocStatement: "DEFPROC" <IDENTIFIER> symbolList (statement)+ "ENDPROC"

symbolList:       "(" (<IDENTIFIER> ("," <IDENTIFIER>)*)? ")"

helpStatement:    "HELP" "(" <IDENTIFIER> ")"
                  | "HELP" <IDENTIFIER>
```

Figure 3-5: Grammar for DEFFUNC, DEFPROC, CALL, and HELP

function	description
strcat	Concatenate any number of strings
directory	Return a URL with the file name removed
dirparent	Return the parent of the given directory
value_id	Return the **value_id** associated with a string. If no string is provided, return a number one higher than the highest **value_id**
value	Return the string associated with a **value_id**
url_id	Return the **url_id** associated with a string or **value_id**
url	Return the url associated with a **url_id**

Table 3.1: User-Callable Functions Defined at Squeal Start-Up

3.3.2 Function/procedure statements

Squeal supports built-in and user-defined *functions* and *procedures*. The body of a function is an expression, and the body of a procedure is one or more statements. The **deffunc** and **defproc** statements are used to declare functions and procedures, respectively, and they are called with the **call** statement. The **help** statement prints information about a built-in function. The grammar for these statements appear in Figure 3-5. Figure 3-6 shows a transcript demonstrating their use. Table 3.1 shows a list of functions defined when the system begins.

3.3.3 Control statements

The grammar for the control statements is shown in Figure 3-7. The INPUT statement specifies from where the Squeal interpreter should receive its input. If the expression is a file name, commands will be read from that file. If the expression is a number, commands will be read from the associated port. The OUTPUT statement indicates that ordinary output should be written to the associated file.

The QUIT statement closes the current input stream to the interpreter, ending the effect of the last INPUT statement. If QUIT is entered at the top input level, the interpreter is exited.

help strcat;
Concatenate any number of strings

? strcat("foo", "bar");
foobar
[printed]

deffunc double(x) strcat(x,x);
[defined function 'double']

? double('hello');
hellohello
[printed]

```
defproc sumdiff(a, b)
    ? a + b;
    ? a - b;
endproc;
```
[defined procedure 'sumdiff']

sumdiff(10,2);
12
8
[printed]

Figure 3-6: Transcript Demonstrating Squeal Functions and Procedures. Commands are in bold face. All other text, including that in brackets, is generated by the Squeal interpreter.

```
inputStatement:  "INPUT" expression

outputStatement: "OUTPUT" stringLiteral

quitStatement:   "QUIT" | "EXIT"
```

Figure 3-7: Grammar for INPUT, OUTPUT, and QUIT statements

table	defining columns	optional columns
link	**source_url_id, anchor_value, dest_url_id**	
page	**url_id**	
parse	**url_value_id**	
rcontains	**value**	**helper, num**
rlink	**anchor_value, dest_url_id**	**helper, num**
tag	**url_id**	

Table 3.2: Defining Columns for Tables

User query:

SELECT * FROM **link** WHERE source_url_id = 1

Normalized form:

SELECT $\mathcal{L}$.* FROM **link** $\mathcal{L}$ WHERE $\mathcal{L}$.source_url_id = 1

Squeal interpretation:

FETCH **link**(source_url_id = 1)

MSELECT * FROM **link** WHERE source_url_id = 1

Figure 3-8: Transformation of Squeal User Query into Internal Statements

3.4 Internal Statements: FETCH and MSELECT

In addition to the Squeal user commands, there are two internal commands that are useful to the Squeal interpreter: FETCH and MSELECT. FETCH does whatever is necessary to fill in lines of a table, given a *defining column* and a value, which can be used to define entire lines of the relation. For example, **source_url_id** is a defining column of **link** because, given a **source_url_id**, all of the values of associated lines can be computed. Table 3.2 shows the defining columns of each of the applicable tables, as well as columns that may optionally be defined in the case of the **rcontains** and **rlink** relations. The **helper** field specifies which search engine should be consulted, and **num** indicates the number of matching pages that should be retrieved. By default, this value is 10. MSELECT, standing for "manual select", simply passes a SELECT statement to the underlying SQL server. As the example in Figure 3-8 shows, SELECT statements are normalized and then broken down into a combination of FETCH and MSELECT statements. This process is discussed in Section 3.5.1.

The FETCH statement

While the Squeal interpreter can be configured to allow the user to enter FETCH and MSELECT statements, they are meant for internal Squeal use. The grammar of the FETCH statement is shown in Figure 3-9. We will consider three separate classes of FETCH statements. During this discussion, it is important to distinguish between *table types*, such as **link**, and *table aliases*, such as $\mathcal{L}$, as illustrated in Figure 3-8. If a query involves only one table alias, the user may omit it, in which case an alias is inserted during Squeal's query normalization process.

Basic FETCH statement The simplest type of FETCH statement does not include a "FROM" clause. How it is interpreted depends on what relation and columns are specified. Figure 3-10 shows how they are interpreted. If multiple columns are supplied, only the earliest one in the relation definition is used. For example,

FETCH **link**(**source_url_id**=1, **dest_url**=2)

is treated as:

```
fetchStatement  = "FETCH" <IDENTIFIER> "(" fetchList ")"
                  ("FROM" tableList
                  ("WHERE" logicExpression)?
                  ("GROUP BY" columnList)?
                  ("HAVING" logicExpression?)?

fetchList       = namedArgument ("," namedArgument)*

namedArgument   = <IDENTIFIER> "=" fetchExpression

fetchExpression = expression "|"expression
                | expression "&" expression
                | expression
```

Figure 3-9: Grammar for FETCH Statement. Nonterminals `columnList` and `orderList` are defined in Figure 3-11. Nonterminals `logicExpression` and `expression` are defined in Appendix B.

table	column
link	**anchor_value**
link	**dest_url_id**
rcontains	**value**
rlink	**anchor_value**
rlink	**dest_url_id**

Table 3.3: Relations Allowing FETCH Conjunctions or Disjunctions

FETCH **link**(**source_url_id**=1)

FETCH **statement with Conjunction or Disjunction** In addition to providing a single expression for a column, it is possible to provide a conjunction or disjunction. For example, consider the following statement:

FETCH **rcontains**(value='Golden'&'Spertus')

This asks Alta Vista for all pages that contain *both* of the words "Golden" and "Spertus" (of which there are 20). Similarly, consider the following statement:

FETCH **rcontains**(value='Golden'|'Spertus')

This finds pages that Alta Vista claims contain *either* "Golden" or "Spertus" (of which there are about 3000). Table 3.3 shows the columns that can be set to conjunctions or disjunctions.

FETCH **statement with FROM clause** To use the contents of a table column as part of the expression, one can supply a FROM clause to a FETCH statement. For example, consider the following FETCH statement, which references user-defined table "utable":

FETCH **link**(**source_url_id** = **url_id**) FROM utable WHERE **url_id** > 10

This is interpreted as:

For all values u of **url_id** in table **utable** such that $u > 10$,
FETCH **link**(source_url_id = u).

table

- **source_url_id**: fetch the page and parse links
- **anchor_value**: perform FETCH rlink(anchor_value=<anchor_value>)), then verify links
- **dest_url_id**: perform FETCH rlink(dest_url_id=<dest_url_id>), then verify links

page

- **url_id**: FETCH page from the Web

parse

- **url_value_id**: parse associated string

rcontains

- **value**: ask search engine for pages containing <value>

rlink

- **anchor_value**: ask search engine for pages with the associated string as anchor text
- **dest_url_id**: ask search engine for pages pointing to <dest_url_id>

tag

- **url_id**: FETCH page from Web or **page** table and parse

Figure 3-10: Interpretation of Simple FETCH Statements. When a search engine is used, it is by default Alta Vista with 10 responses retrieved, but these parameters can be changed via the **helper** and **num** columns, respectively.

nonterminal	statement	description
createStatement	CREATE	create a table
dropStatement	DROP	drop (delete) a table
describeStatement	DESCRIBE	describe an existing table
insertStatement	INSERT	insert rows into a table
updateStatement	UPDATE	change values in a table
deleteStatement	DELETE	delete rows from a table
selectStatement	SELECT	select rows from a table

Table 3.4: SQL Commands Supported by Squeal

```
selectStatement = ("SELECT"|"MSELECT") ("ALL" | "DISTINCT") ? selectList
                  "FROM" tableList
                  ("WHERE" logicExpression)?
                  ("GROUP BY" columnList)?
                  ("HAVING" logicExpression)?
                  ("ORDER BY" orderList)?

selectList      = selectItem ("," selectItem)*
selectItem      = expression ("AS" <IDENTIFIER>)?

tableList       = tableName ("," tableName)*
tableName       = <IDENTIFIER> (<IDENTIFIER>)?

orderList       = orderItem ("," orderItem)*
orderItem       = expression ("ASC" | "DESC")
columnList      = column ("," column)*
column          = <IDENTIFIER> ("." <IDENTIFIER>)
```

Figure 3-11: Grammar for Squeal Queries. The definitions of nonterminals `logicExpression` and `expression` appear in appendix B.

3.5 Squeal's SQL core

Table 3.4 lists the SQL commands supported by Squeal. None of them except for SELECT can be applied to the system tables described in chapter 2, only to user-defined tables. By default, the CREATE statement does not cause an error if the table being defined already exists; instead, it drops the old table and then creates a new one. SQL-compliant CREATE behavior can be achieved with the "-c" command-line option 3.6. Figure 3-11 shows the simplified grammar for SELECT statements.

As discussed above, not all syntactically-valid statements are acceptable in Squeal. Tables are divided into three categories, *user-readable*, *automatic*, and *derived*, as shown in Table 3.5. Different rules apply for queries depending on their category.

3.5.1 User-Readable Tables

User-readable tables include **urls** and **valstring**, as well as any tables created by the user. Squeal passes queries on these tables directly to the SQL client, without performing and checks or side effects. Changes are effected to **urls** through the **url_id** function (section 2.2.1), to **valstring** through the **value_id** function (section 2.2.1), and to user-defined tables through INSERT statements.

table	category
urls	user-readable
valstring	user-readable
link	automatic
page	automatic
parse	automatic
rcontains	automatic
rlink	automatic
tag	automatic
att	derived
header	derived
list	derived

Table 3.5: Categories of Tables. Each table is either *user-readable*, *automatic*, or *derived*. User-defined tables are user-readable.

3.5.2 Automatic Tables

Queries on automatic tables, such as **link** require analysis by the Squeal interpreter, which will turn them into FETCH and MSELECT statements. They are called "automatic" because the entries are automatically filled in response to user queries. Each automatic table has one or more *defining columns*, as discussed in section 3.4.

The most basic query involving an automatic table is of the form:

```
SELECT * FROM <table type> <table alias> WHERE (<col name> = <expression>)
```

If <col name> is a defining column for <table type>, this is transformed into:

```
FETCH <table type>( <col name> = <expression> )
MSELECT * FROM <table type> WHERE (<col name> = <expression>)
```

If <col name> is not a defining column, the interpreter rejects the statement.

The four key procedures involved in generating FETCH statements from SELECT statements are:

1. **transform**: top-level routine, which calls the below procedures and outputs required FETCH statements (Figure 3-12).
   (a) **merge**: recognize a conjunction idiom, possibly outputting a FETCH statement (Figure 3-14).
   (b) **findBound**: try to find a bound on a defining column of an automatic table appearing in the SELECT statement; may call itself recursively (Figure 3-13).
   (c) **refDependences**: help construct FETCH statements, adding references to tables upon which the current table depends; may call itself recursively (Figure 3-15).

3.5.3 Derived Tables

Derived tables are not computed directly. Instead, they are computed when their *parent table* is computed. For example, **header** for a **source_url_id** is computed as a side effect of computing **tag** for the same **source_url_id**. Table 3.6 shows the parent tables of each of the derived tables and the column they have in common with their parent.

Derived tables are dealt with similarly, but not identically, to automatic tables. All changes are to procedure **findBound**. Figure 3-16 shows the changes required to **findBound**.

Procedure **transform**(selectStatement)

1. Initialization
   (a) Let unboundedTables be the set of table aliases in selectStatement.
   (b) Let whereDef be the WHERE clause in selectStatement.
   (c) Let selectionTable by an empty hash table indexed on table aliases.
   (d) Let orderedKeys be an empty vector.
2. Call merge(selectStatement, unboundedTables)
3. Let remainingTables be |unboundedTables|.
4. For each table alias currentTableAlias in unboundedTables,
   call findBound(currentTable, unboundedTables, fetchClauses, whereDef)
5. Branch point on the value of |unboundedTables|:
   (a) remainingTables, fail.
   (b) > 0, go to step 3.
   (c) else, execute completion code:
      i. For each alias in orderedKeys, if the table is not a UserReadableTable
         A. Let fetchString = "FETCH ".
         B. Let tablesString = "".
         C. Let clausesString = " WHERE (1=1) "
         D. Let dependences be the empty set
         E. For each tuple [currentTableAlias, LHSstring, op, RHSstring, tablesReferenced] associated with alias in selectionTable where LHSstring $\neq$ the empty string
            1. Append to fetchString: LHSstring + op + RHSstring
            2. Add the elements of tablesReferenced to dependences.
            3. Let processed be the set containing alias
            4. call refDependences(alias, selectionTable, dependences, tablesString, clausesString, processed)
            5. Output clause: fetchString + ") FROM " + tablesString + clausesString
      ii. return success

Figure 3-12: Pseudocode for **transform**. Given a selectStatement, **transform** returns a set of clauses or fails. Code to handle simple syntactic items, such as commas, has been omitted.

derived table	parent table	shared column
att	tag	tag_id
header	tag	url_id
list	tag	url_id

Table 3.6: Relations Between Derived Tables and Their Parents

Procedure **findBound**(currentTableAlias, unboundedTables, fetchClauses, whereDef)
Branch point on the type of whereDef

1. disjunction (clause1 OR clause2)
   (a) Let returnValue1 be the result of findBound(currentTableAlias, unboundedTables, fetchClauses, clause1)
   (b) If returnValue = false, return false
   (c) Let returnValue2 be the result of findBound(currentTableAlias, unboundedTables, fetchClauses, clause2)
   (d) Merge any new clauses of fetchClauses with the same left-hand side, to create FETCH disjunctions (section 3.4).
   (e) Return returnValue2
2. conjunction (clause1 AND clause2)
   (a) Let returnValue1 be the result of findBound(currentTableAlias, unboundedTables, fetchClauses, clause1)
   (b) Let returnValue2 be the result of findBound(currentTableAlias, unboundedTables, fetchClauses, clause2)
   (c) Merge any new clauses of fetchClauses with the same left-hand side, to create FETCH conjunctions 3.4.
   (d) return (returnValue1 $\vee$ returnValue2)
3. negation (e.g, "tab.col $\neq$ 7"), return false
4. relational expression (LHS op RHS)
   (a) If (currentTableType is not a UserReadableTable and (*op* is not "=" or "LIKE") or
   (*LHS* is not of the form currentTableAlias.col) or
   (col is not a defining column for currentTableType)),
   return false
   (b) Let currentTableType be the type of currentTableAlias
   (c) Let referencedTables be the tables referenced in *RHS*
   (d) If |referencedTables $\cap$ unboundedTables|$\neq$ 0,
   return false
   (e) Let newClause be the tuple
   [currentTableAlias, ToString(LHS), op, ToString(RHS), tablesReferenced]
   (f) Add newClauses to fetchClauses
   (g) Remove currentTableAlias from unboundedTables
   (h) return true

Figure 3-13: Pseudocode for **findBound**. Returns true if an item was removed from unboundedTables, false otherwise.

Procedure **merge**(selectStatement, unboundedTables)

Look for WHERE clauses of the form:

t1.col1 = t2.col1 AND
t1.col2 = exp1 AND t1.col2 = exp2

where *t1* and *t2* are different instantiations of the same table T. If found, remove *t1* and *t2* from unboundedTables and execute:

FETCH T(col2 = (exp1 & exp2))

Figure 3-14: Description of **merge**

Procedure **refDependences**(topAlias, selectionTable, tablesReferenced, tablesString, clausesString, processed)

1. Let newDependences be the empty set
2. For each alias predAlias in tablesReferenced and selectionTable but not in processed
   (a) Put predAlias in processed.
   (b) Append to tablesString: TableType(currentTableAlias) + " " + currentTableAlias
      i. Append to tablesString: TableType(predAlias) + " " + predAlias
      ii. For each tuple [currentTableAlias, LHSstring, operator, RHSstring, currentTablesReferenced] corresponding to predAlias in selectionTable where currentTablesReferenced does not contain topAlias,
         A. Append to clausesString: " AND " + LHSstring + " " + operator + " " + RHSstring
         B. Add the elements of currentTablesReferenced to the set newDependences
3. If | newDependences |> 0, call refDependences(topAlias, selectionTable, tablesReferenced, tablesString, clausesString, processed)

Figure 3-15: Pseudocode for **refDependences**. The procedure adds table definitions and clauses of referenced tables to the current FETCH statement.

4. if whereDef is a relational expression (LHS op RHS)

   (a) If (currentTableType is not a UserReadableTable and (*op* is not "=" or "LIKE") or
   (*LHS* is not of the form currentTableAlias.col) or
   (col is not a defining column for currentTableType)),
   return false

   (b) Let currentTableType be the type of currentTableAlias

   (c) Let parentTable be the table from which currentTableType is derived

   (d) Let parentCol be the column through which it is derived

   (e) If *RHS* is parentTable.parentCol where parentTable is the parent table of currentTableType and parentCol is the column through which they are related,
   return true

   (f) If |referencedTables ∩ unboundedTables|≠ 0,
   return false

   (g) If *RHS* is a constant expression

      i. Let newClause be the tuple
      [parentTable, parentCol, ToString(RHS), tablesReferenced]

      ii. Add newClauses to fetchClauses

      iii. Remove currentTableAlias from unboundedTables

      iv. return true

   (h) return false

Figure 3-16: Changed Portion of **findBound** (Figure 3-13) for Derived Tables

```
usage: java FrontEnd [-d] [-log <filename>] [-maxpage #] [-tolinks #] <filename>*
        -d: debug
        -c: creation of tables that already exist illegal
        -log <filename>: output detailed information to a file
        -maxpage #: Maximum size (in Kbytes) of pages to load (default is 30)
        -tolinks #: Minimum number of pages to consider per query (default is 10)
```

Figure 3-17: Usage for Squeal

3.6 Command-Line Interface

Figure 3-17 shows the command-line interface to Squeal. If the "debug" (-d) flag is set, extra information is printed to the standard output stream. If the "creation" (-c) flag is set, it is illegal to redefine a table that already exists; otherwise, the new definition replaces the old one. If the "log" option is used, detailed debugging and status information is written to a file. The variable "maxpage" limits the size of pages to load; by default, the limit is 30 kilobytes. The variable "tolinks" controls how many links should be retrieved from a search engine after making a query; the default is ten. Input files can be provided on the command line.