BRL

A database-oriented language to embed in HTML and other markup
Release 2.1.21, 1 December 2000

Bruce R. Lewis
Eaton Vance Management

1 Introduction

BRL is a language designed for server-side WWW-based applications, particularly database applications. Its facility for generating output from databases makes it a *report* language. It hides powerful semantics behind a simple syntax, resulting in code that is easy to write, read and maintain. This makes it a *beautiful* language. Thus its name, BRL, the *Beautiful Report Language*.

BRL programs are very much like web pages. One simply puts a BRL file on a HTTP server and calls up the appropriate URL In a web browser. The first time the page is loaded it is compiled, and the same compiled program is used for subsequent page loads until the underlying BRL file is modified.

There are other systems besides BRL that work in a similar manner, but they usually require learning a programming language that is unique to that system (e.g. PHP, CFML), or a language that is more cumbersome than necessary (e.g. Java/JSP) for the simple programming usually demanded in HTML pages. BRL uses Scheme, a language taught in hundreds of universities, colleges and secondary schools worldwide.

Scheme uses an extremely uncomplicated syntax that makes simple code look simple. It is popular for teaching Computer Science (CS) because instructors can spend a short time teaching the language itself, leaving more time to teach CS principles. This has given Scheme a reputation for being a difficult language because it is usually associated with advanced concepts. However, those advanced concepts are not necessary for writing applications using BRL.

BRL is suitable not just for HTML, but for any markup language. A combination of BRL and pdflatex can be used to dynamically generate PDF files.[1]

BRL uses a database-neutral SQL interface, borrowed from Java's JDBC. The current implementation of BRL is a Java Servlet, allowing integration with all the most popular HTTP servers. All major operating system / database / HTTP server combinations are suitable for running BRL.

1.1 About This Manual

This manual gives you enough information to use BRL to write sophisticated web applications. The "BRL by Example" chapter illustrates the greatest strengths of BRL, but the "BRL Reference" section describes many more capabilities.

BRL is stable and fast enough for production use, but is still a work in progress. If there are features you particularly would like to see, please let the author know. This will help prioritize future development: *brlewis@users.sourceforge.net*

1.1.1 Further Help

If at any point in this manual you need further help, ask on Usenet. If you don't know what Usenet is, see Harley Hahn's What Is Usenet?

[1] This functionality is not yet documented, but has been done using `brl-handle-request` and Kawa's `system` procedure.

Different sections in this manual will mention different Usenet groups to consult. But if no other group is mentioned, BRL questions can be directed to '`comp.lang.scheme`', and comments to '`comp.infosystems.www.databases`'. Be sure to put BRL in the subject line to make sure your posting is not overlooked.

If you did not see a URL in parentheses after "What Is Usenet" above, you are probably looking at a printout of the PDF version of this manual. Look at the PDF version online to be able to follow the hyperlinks.

For those who cannot or do not want to post on Usenet, there are also BRL mailing lists.

2 Downloading and Installing BRL

2.1 Prerequisites

Before you can run BRL, you need the following:

1. A servlet engine that supports the Servlet API 2.0 or later.
2. Release 1.6.70 of Kawa

Your servlet engine should come with '`SnoopServlet.class`' or some other little test servlet. Do not try to install BRL until you've successfully tested such a servlet.

If you need further help, consult one of the following Usenet groups according to what OS your web server runs on:

- '`comp.infosystems.www.servers.unix`'
- '`comp.infosystems.www.servers.mac`'
- '`comp.infosystems.www.servers.ms-windows`'
- '`comp.infosystems.www.servers.misc`'

The rest of these instructions assume you already have servlets working. Consult your servlet engine's documentation for details. With JServ on Debian GNU/Linux, I had to add the following lines to httpd.conf to get things working.

```
<IfModule mod_jserv.c>
  ApJServLogFile /var/log/jserv.log
  Include /etc/jserv/jserv.conf
</IfModule>
```

2.2 Downloading BRL

You can get a compiled jar file or tar-gzipped sources from BRL's SourceForge Project Page, or via anonymous FTP.

2.3 Easy Installation

1. Put the compiled jar file, '`brl-2.1.21-compiled.zip`' into the `CLASSPATH` of your servlet engine, as well as the compiled jar file for Kawa. For example, if you are using JServ, add lines like the following to '`jserv.properties`':

   ```
   wrapper.classpath=/usr/local/share/java/brl-2.1.21.jar
   wrapper.classpath=/usr/local/share/java/kawa-1.6.70.jar
   ```

2. Extract '`brl-global.scm`' and '`sitedefs.scm`' from the zip file and put them in a directory accessible to your servlet engine, but preferably not accessible to outsiders, as you might later put database passwords in '`sitedefs.scm`'.
3. If your servlet engine allows it, configure the servlet `gnu.brl.brlsv` to be run for pages ending in '`.brl`'. For example, if you are using JServ, add the following directive to '`jserv.conf`':

   ```
   ApJServAction .brl /servlets/gnu.brl.brlsv
   ```

   Otherwise extract brlsv.class from the zip file and copy it to your servlets directory.

4. Set the servlet initialization parameter `scmdir` to the directory you put the '`.scm`' files in. For example, if you are using Jserv, add the following directive to '`root.properties`':

   ```
   servlet.gnu.brl.brlsv.initArgs=scmdir=/usr/local/com/brl
   ```

   If you would rather specify a location relative to the web root, you can set the servlet initialization parameter `scmuri` to such a path, e.g. `/WEB-INF`.

   By default the BRL servlet expects .brl files to be under your web server's document root. Some sites may prefer to put them in a different directory. Such sites should set the servlet initialization parameter `brldir` to the full path of that directory.

2.4 Building from Source

You need a Java compiler and a Unix-like make utility such as gmake. Issue the following shell command from the directory that contains a file called configure:

```
./configure
```

Then choose one of the following two commands for install:

If you want a zip file to put in your class path:

```
make brl-compiled.zip
```

Note that the resultant '`.zip`' file has a name that includes the current version number, e.g. '`brl-2.1.21-compiled.zip`'. Follow the "Easy Installation" instructions above.

If you want to install individual class files in '`/usr/local/share/java`':

```
make install
```

This method will also keep you from having to set the scmdir servlet parameter.

2.5 Testing Your Installation

Let's say your HTTP server is called `example.com`. If you have configured your servlet engine to run `gnu.brl.brlsv` for pages ending in '`.brl`', go to this URL:

```
http://example.com/noexist.brl
```

Otherwise go to this URL:

```
http://example.com/servlet/gnu.brl.brlsv/noexist.brl
```

If you get the usual "404 Not Found" error, then you have not successfully configured your servlet engine to run BRL. Go back to the documentation for your servlet engine. If you have correctly installed and configured BRL, then you should see a message with "Error" and "Debugging Info" sections, and "brl servlet" with a release number at the bottom.

If the error message is "gnu/expr/ModuleBody", then your servlet engine's class loader cannot handle the standard version of Kawa. Look for the `kawa-brl` package on BRL's SourceForge Project Page.

The error message should say, "File not found". You should be able to tell from this error message exactly where the BRL servlet is trying to find files. Now you are ready to begin using BRL.

3 Introducing Scheme

It's fairly easy to pick up basic BRL syntax just by looking at examples. Nonetheless, it's helpful to understand what's happening behind the scenes if you want to understand the subtleties of an example, or if you want to go beyond the things you already have examples for.

The powerful language behind BRL is Scheme. This language was designed for teaching, and there is a very friendly tool available for learning it called DrScheme.

If you're using the machine you installed BRL on, you can start Kawa Scheme from the shell or command prompt by typing this:

```
java kawa.repl
```

Kawa Scheme will let you try Scheme/Java integration, but won't provide as friendly an environment as DrScheme.

3.1 Simple Expressions

We'll begin, as is customary in introductions to programming languages, with the syntax for printing the words "Hello, World" on the screen:

```
> "Hello, World"
Hello, World
```

The `>` sign represents the prompt at which you type. It might be `#|kawa:1|#` or something else depending on the scheme implementation. But the result is the same. You type "Hello, World" (*including* the quotes) and you get "Hello, World". This may be a little confusing if you're used to other computer languages in which "Hello World" can only be used as an argument to a command. In Scheme you use the term *expression* for both concepts (command and argument). Everything you type in a Scheme interpreter is an expression. "Hello, World" is a what's called a *self-evaluating expression*, or a *constant*.

Here's how you assign expressions to variables:

```
> (define h "Hello, World")
> h
Hello, World
```

You may have used other languages in which variables have a special character to mark them, e.g. `$h` or `@h`. In Scheme variable names are simple. A variable name is an expression that evaluates to whatever you assigned to it. The `define` expression above is an example of variable *assignment*, more commonly called *binding* in Scheme. The variable name is said to be *bound* to a value.

Continuing the above example, the following is an expression which Scheme evaluates as follows: The variable `string-append` evaluates to a procedure. The variable `h` evaluates to "Hello, World", and the string "!!" evaluates to itself. Then `string-append` is applied to its two *arguments* "Hello, World" and "!!", resulting in one string formed from the two.

```
> (string-append h "!!")
Hello, World!!
```

As you can see, this is a very simple process, called *procedure application*. Don't be intimidated by the use of terms like *procedure*, *arguments*, *variable*, *binding*, and *self-evaluating expression* to describe it. I only introduce those terms here to help you read Scheme documentation later.

Not every Scheme expression works by procedure application. You might have guessed this, as the `(define h "Hello, World")` expression could not have worked by first evaluating h and passing its value as an argument to define. At that point, h was not defined yet and could not evaluate to anything.

There are other examples in Scheme of expressions that are not procedure application, but in general the syntax is the same: First an open parenthesis, then what you're doing, then what you're doing it to, then a close parenthesis. This can be a little confusing at first for math if you're not used to it:

```
(+ (* 2 3 4) 5 6)
```

The normal math notation would be 2x3x4 + 5 + 6. You will find it less confusing if you read `+` in Scheme code as "the sum of" and `*` as "the product of".

3.2 Defining Procedures

One of Scheme's great strengths is how simple it is to extend with new functions. Here's a simple example:

```
> (define (greet name) (string-append "Hello, " name "!!"))
> (greet "Bruce")
Hello, Bruce!!
```

Note that there are *two* parentheses at the end of the definition. One closes the `string-append` expression, while the other closes `define`. Scheme code is usually written in an editor such that when you type a close paren, the corresponding open paren is highlighted. Also, since it is parens and not line breaks that end a Scheme expression, definitions like the one above are often split into multiple lines with indentation clarifying how "deep" the code is within parentheses.

Your procedure can be used anywhere you would normally put a Scheme expression:

```
> (string-append "Wow!  " (greet "Chris") "  Long time, eh?")
Wow!  Hello, Chris!!  Long time, eh?
```

You can use your function to define other functions:

```
> (define (safe-greet name)
    (if (string? name)
        (greet name)
        (greet "NON-STRING!")))
> (safe-greet 2)
```

Also demonstrated in the above example is the `if` syntax. If the first argument is not false, then the second argument is evaluated, otherwise the third argument is evaluated.

Here's a better example of `if`:

```
> (define (price n)
    (string-append "just "
                   (number->string n)
                   (if (= 1 n) "dollar" "dollars")))
```

This demonstrates that an `if` expression can go anywhere. This may seem obvious if you're new to programming, but there are many languages in which `if` is used more restrictively. People accustomed to such languages may be surprised to see an `if` expression used as an argument to a procedure. As an exercise, rewrite `safe-greet` so that `greet` appears only once in its definition. Do this by making the argument to `greet` be an `if` expression.

3.3 Data Structures

NOTE: If you're using DrScheme, you'll need to set the language to "Full Scheme" to run the examples in this section.

This section will describe all but the least-used data structures in Scheme. To start, let's look at the simplest structure: a *pair*. This structure groups two items together. The items are retrieved with the `car` and `cdr` procedures:

```
> (define c (cons 1 2))
> (car c)
1
> (cdr c)
2
```

Try doing something similar with strings instead of numbers, but do it without looking at the above example. Memorize what `cons`, `car` and `cdr` do.

Did you do it? Good, we're done covering all but the least-used data structures in Scheme. But wait! We've only covered one very simple data structure. How can I say we've covered so much?

What makes pairs so powerful is that *any* data type can be stored in them, *including other pairs*. This allows one to string together (or, as the LISP community likes to say, *cons up*) a list by putting the first element of the list in the `car` and another pair in the `cdr`. This second pair has the second list element as its `car` and the rest of the list as its `cdr`, and so on. Eventually you reach the end of the list, where the `cdr` is a special marker known as the empty list.

```
> (define nums (list 1 2 3))
> (car nums)
1
> (cdr nums)
(2 3)
```

Scheme includes many tools for list processing. I will demonstrate two of them here.

```
> (define (square x) (* x x))
> (map square nums)
(1 4 9)
> (apply + (map square nums))
14
```

As you can see, `map` applies a procedure to each element of a list and returns a list of the results. In this example, `square` was applied to 1, 2 and 3, resulting in 1, 4 and 9, respectively. The `apply` procedure was used to call the `+` procedure with 1, 4 and 9 as arguments, returning 14.

The uses of `map` and `apply` are limitless. For example, if you had a procedure that computed sales tax, you could use `map` to get a list of sales taxes from a list of prices, and then use `apply` to total them. But the great thing about Scheme is not the existence of generally-useful procedures like `map` and `apply`, but how easily you can create such generally-useful procedures yourself.

For example, suppose you wanted to quiz yourself on arithmetic. Whether you're doing addition, multiplication, division or subtraction, the basic idea is the same. You take two numbers, perform an operation on them, and compare that to input. If the correct answer was input, say "Correct!", otherwise give the correct answer.

```
> (define (quiz op n1 n2)
    (display "Your answer? ")
    (if (equal? (op n1 n2) (read))
        "Correct!"
        (op n1 n2)))
> (quiz * 2 3)
Your answer? 6
Correct!
```

In other languages you would have had to write separate pieces of your program for the various arithmetic operations. But with Scheme you've written one general-purpose procedure that works with all of them, plus operations that you didn't even know existed. Try it with `remainder`, `lcm` and `gcd`. Not bad for a five-line program, eh?

3.4 Learning More

It's only fair to warn you that there are few resources out there just for learning Scheme. There are plenty of books that might seem to be presenting Scheme, but they are actually teaching Computer Science, and expecting you to learn Scheme as you go along. Your best approach if you just want to use BRL is to take one of these books and skim through anything that seems difficult to understand, focusing on the examples.

Computer Science principles will be very useful to you if you tackle a large software project, but for the most common use of BRL, i.e. web pages, it's enough just to know the basics of Scheme.

Another good way to learn Scheme is to look at the specification, R5RS. This document is really targeted at people making programs that interpret or compile Scheme, not those who simply want to program in Scheme. It is rather dense in places, and some of the examples are tricky because they demonstrate several principles at once. However, you can still find it a useful introduction if you don't let yourself get bogged down.

A third option is to simply keep reading this manual and learn from example, then go to R5RS or a textbook later.

4 Learning BRL by Example

Once you've learned some Scheme, it's trivial to create BRL web pages. Simply enclose Scheme expressions in square brackets wherever you want the resulting value to appear on the web page. If you want a literal open bracket, put two. You don't need to do anything special for a close bracket.

If you put multiple expressions within a set of square brackets, the resulting values (if applicable) will appear with no whitespace in between.

Conceptually, BRL is no different from the interactive Scheme interpreter in which you typed "Hello, World" in the last chapter. The only difference is that literal strings can be found outside square brackets in addition to being found inside quotes. Everything in a BRL page is treated as a Scheme expression, and the result of each expression is output in sequence to form an HTML page.

4.1 HTML Forms and CGI Environment Variables

```
 1 <html>
 2 <head>
 3 [
 4  (inputs word) ; HTML input.  Will be null if no such input.
 5  (define newword
 6    (if (null? word)
 7        "something"
 8        word))
 9 ]
10 <title>Backwards</title>
11 </head>
12
13 <body>
14 <form>
15 Type a word: <input name="word">
16 <input type="Submit">
17 </form>
18
19 <p>[newword] spelled backwards is
20    [(list->string (reverse (string->list newword)))]
21 </p>
22
23 <p>This message brought to you by [(cgi SERVER_NAME)] as a public
24 service.</p>
25
26 </body>
27 </html>
```

In this example, the `inputs` syntax on line 4 is used to name a variable that will hold the value of an HTML input of the same name. Here's something important to remember about HTML forms: You never know what value you're going to get, or if you're going to get any input at all. **Always validate your inputs.** In this example, `newword` is defined in

line 5 so that we are guaranteed to have a string to work with, even when no `word` input is supplied.

Note that if `word` is supplied but is an empty string, `(null? word)` will be *false*. If you want to treat blank or unsupplied inputs the same way, you could use `brl-blank?` in place of `null?` or use `brl-nonblank?` in the reverse sense.

Continuing with the above example, lines 10-18 are straight HTML to provide a form that refers back to this same page. It's important to note that the HTML input on line 15 must have the same name as the BRL input on line 4.

On line 19, `[newword]` shows where the value of the `newword` variable goes. Then on line 20 we have a string formed from a reversed list of the characters in `newword`. On line 23 we see BRL's `cgi` syntax, used to get CGI environment variables

4.2 SQL in BRL

4.2.1 One-Time Preparation for All Pages

The following examples will assume you have a JDBC driver for a database in which you've created the following table:

```
create table favcolor(
    name varchar(20),
    color varchar(20))
```

In your '`sitedefs.scm`' file, register your JDBC driver with the `sql-driver` procedure. Then define a procedure for connecting to your database. By defining this procedure in your '`sitedefs.scm`' file, you avoid putting database usernames/passwords in BRL web pages. For example, if you were using the freetds driver:

```
(sql-driver "com.internetcds.jdbc.tds.Driver")
(define (db1 brl-context)
  (brl-sql-connection brl-context "jdbc:url-here" "user-here"
                      "password-here"))

(define ss brl-sql-string)
```

The `brl-sql-connection` procedure returns a connection to your database that can then be used to create SQL statements, and assures that this connection will be closed once the BRL page finishes, whether normally or by error. The `brl-context` variable is a structure that contains various information in the context of the current BRL page.

The purpose of `(define ss brl-sql-string)` in this context is in case you switch back and forth between a driver that expects standard SQL string syntax (only single quotes need escaping) and a driver that expects non-standard string syntax, e.g. MySQL (backslashes also need escaping). Use `ss` wherever you would otherwise use `brl-sql-string` or `brl-mysql-string`.

If you need further help getting your driver working, consult your JDBC driver supplier or the following Usenet group: '`comp.lang.java.databases`'

4.2.2 Queries in an Individual Page

```
1  [
2   (define conn (db1 brl-context))
3   (define st (brl-sql-statement brl-context conn))
4  ]<html>
5  <head><title>Favorite Colors</title></head>
6  <body>
7
8  <ul>
9  [(define rowcount
10    (sql-repeat st (name color) ("select * from favcolor")
11      (brl ]<li> [name] likes [color]
12  ]))]
13 </ul>
14
15 <p>Count: [rowcount]</p>
16
17 </body>
18 </html>
```

In line 2, the `db1` procedure previously defined in '`sitedefs.scm`' is used to get an SQL connection that will be automatically closed later. In line 3, the `brl-sql-statement` procedure is used to get an object that can be used repeatedly for SQL queries on that connection. It is currently implemented as JDBC's `java.sql.Statement` object.

In line 10, we see BRL's `sql-repeat` syntax. The first argument is the SQL statement object to be used in the query. The next argument specifies the variable names to be used for the columns returned by the query, in this case `name` and `color`. The third argument is the text of the query. In this example it is just one string, but it could have been several pieces, some strings, some not, that would be concatenated.

Any remaining arguments to `sql-repeat` are Scheme expressions to be evaluated once for each row returned in the query. In this example, there is just one expression. In standard Scheme, the expression would be written as follows:

```
(brl "<li> " name " likes " color "
")
```

BRL's square-bracket syntax helps clarify what text is literal and what is evaluated. It just takes a little getting used to code like `(brl ]`, which signifies the beginning of literal text contained within a BRL expression. This same syntax can be used for SQL queries, as will be seen in later examples.

The `brl` syntax is necessary because of the fundamental difference between BRL and Scheme. In Scheme, sequences of expressions generally return the value of the last expression. In BRL, all expressions in a sequence are output. The `brl` syntax is most commonly used in `sql-repeat` and `if` expressions.

The `sql-repeat` syntax returns the number of rows from the query. Since BRL pages normally show the return value of expressions, this example uses `define rowcount` to capture the value so that it can be output in a more appropriate place on line 15.

The three parentheses on line 12 close the `brl` expression, the `sql-repeat` expression, and the `define` expression, respectively.

4.2.3 Inserts, Updates and Deletes

For people to enter their own favorite colors into the database, they'll need a form. This is simply HTML; there is no BRL programming involved.

```
 1  <html>
 2  <head><title>Choose Favorite Color</title></head>
 3
 4  <body>
 5
 6  <form action="chosen.brl">
 7  Name: <input name="name"><br>
 8  Color: <input name="color"><br>
 9  <input type="submit">
10  </form>
11
12  </body>
13  </html>
```

The `action` in line 6 will work if your web server knows how to handle the '`.brl`' extension. If not, you'll need to use a longer URL. Here is '`chosen.brl`':

```
 1  <html>
 2  <head><title>Favorite Color Chosen</title></head>
 3  [(brl-referer-check brl-context)
 4   (inputs name color)
 5   (define conn (db1 brl-context))
 6   (define st (brl-sql-statement brl-context conn))
 7  ]
 8  <body>
 9
10  [(if (or (brl-blank? name)
11           (brl-blank? color))
12       "<p>Please go back and fill in your name/color.</p>"
13       (if (positive? (sql-execute-update st
14                       "update favcolor set color="
15                       (brl-sql-string color)
16                       " where name="
17                       (brl-sql-string name)))
18           "<p>Your favorite color has been updated.</p>"
19           (begin
20             (sql-execute-update st
21                   ]insert favcolor(name, color)
22  values([(brl-sql-string name)], [(brl-sql-string color)])[)
23             "<p>Welcome to the favorite color database!</p>"
24           )
25       )
26  )]
27
28  </body>
29  </html>
```

You've seen lines like 1-8 before, except for line 3. If it weren't for the `brl-referer-check` expression ("referrer" misspelled for historical reasons), some joker could create a form on his own web site that looks like it's doing one thing, but then when a user hits the Submit button it would change the user's favorite color in *your* database. The `brl-referer-check` expression restricts referers to your own web site. If this is not restrictive enough, you can write your own code to check the value of `(cgi HTTP_REFERER)`. A simple `brl-referer-check` will eliminate most mischief, but if your site includes URLs submitted from outside sources, or if some of your privileged users have setups that do not send referer info, you should include additional checks in your code.

Here is a translations of lines 10-23 into English: If either input has not been filled in (10-11), return a message asking the user to do so (12). Otherwise, try executing an `update` for the user (13-17). If that update affects a positive number of rows (13), return an update message (19). Otherwise insert the appropriate data (19-22) and return a welcome message (23).

The `sql-execute-update` procedure takes two arguments or more. The first is the statement object to use. The second and subsequent arguments become a string representing an SQL insert, update or delete. The return value is the number of rows affected.

Starting on line 14, the standard Scheme syntax is used for strings. The alternate BRL syntax is used starting on line 21, using square brackets to mark the dividing lines between Scheme code and literal strings. Use the standard Scheme syntax for short, simple strings. Use the BRL syntax for longer strings or strings containing lots of double quotes or parentheses, or anywhere that you think someone reading your code would have trouble telling whether a double quote is starting or ending a literal string.

The `brl-sql-string` procedure on lines 15, 17 and 22 takes one argument. If this argument is null, `"NULL"` is returned. Otherwise a string is returned enclosed in single quotes, with any internal single quotes doubled, as required by SQL. This is important. A knowledgeable intruder might otherwise supply form input that executes arbitrary SQL statements in your database. If you are using or might start using MySQL, which has has non-standard string-escaping rules, see Section 4.2.1 [One-Time Preparation for All Pages], page 10.

The final bit of syntax that deserves explanation is in lines 19-24. Use `begin` to group expressions so that only the value of the last expression is returned. If you want all the values to be output, group the expressions with `brl` instead. In this example, we don't want to see the number of rows affected by the insert; that number should always be 1.

4.2.4 Grouping Results

SQL results do not always map neatly into the result you want to see on an HTML page. Sometimes you want subtotals for numeric data or sectional divisions for other data. Like other report systems, BRL provides a mechanism for grouping SQL results. For example, suppose you had the following results:

```
color       name
------      -----
blue        Jane
blue        Joe
blue        Lancelot
```

```
red          Bill
red          Yuri
```

Suppose you wanted output that looked like this:

- **blue**: Jane, Joe, Lancelot
- **red**: Bill, Yuri

Here is the BRL code that lets you do it:

```
 1  [
 2   (define conn (db1 brl-context))
 3   (define st (brl-sql-statement brl-context conn))
 4  ]<html>
 5  <head><title>Favorite Colors</title></head>
 6  <body>
 7
 8  <ul>
 9  [(define rowcount
10     (sql-repeat st (name color)
11  ("select * from favcolor
12   order by color, name")
13    (if (group-beginning? color)
14        (brl ]<li><strong>[color]</strong>: [name)
15        (brl ", " name))
16    (if (group-ending? color)
17        (brl #\newline))))]
18  </ul>
19
20  <p>Count: [rowcount]</p>
21
22  </body>
23  </html>
```

Lines 1-12 aren't significantly different from earlier examples in this chapter. Lines 13-17 are where it gets interesting. The `group-beginning?` syntax determines if a group is beginning. In this example, `color` determines the beginning of the group. So as each color group begins (13), a bullet is output, along with the color and a colon, plus the first name associated with that color (14). If a group is not beginning (i.e. 2nd and subsequent names), a comma is output followed by the name (15). Then, if we are at the end of a group based on color (16), a `#\newline` character is output. This just makes the HTML output cleaner in case you do "View Source" from your browser.

If you want a subgroup based on a column, e.g. `col2` with a group based on `col1`, use `(group-beginning? col1)` to test for the outer group and `(group-beginning? (list col1 col2))` to test for the inner group. To nest groups deeper, simply add the column names to the list.

The argument to `group-beginning?` or `group-ending?` does not have to be just a column name. For example, `(group-beginning? (string-ref color 0))` would group colors together by their first letter.

If you use a constant, `group-beginning?` will be true only at the beginning of the result set, and `group-ending?` will be true only at the end of the result set. I suggest using the symbol `'all` for this purpose, as it makes your intent clear.

4.2.5 URL and HTML Escapes

Continuing with the previous examples, we've set up a web site that lets people put in arbitrary strings as colors. This introduces a security issue: What if someone introduces a "color" that is actually JavaScript code or some such? The code might cause other users' browsers to do something undesired. The easy way to fix this problem is to translate the characters `<`, `>`, `&` and `"` into their respective HTML escape sequences. This is done in the following example with the `brl-html-escape` procedure.

```
 1  [
 2   (define conn (db1 brl-context))
 3   (define st (brl-sql-statement brl-context conn))
 4  ]<html>
 5  <head><title>Choose a Color</title></head>
 6  <body>
 7
 8  <ul>
 9  [(define rowcount
10     (sql-repeat st (color)
11  ("select distinct color from favcolor
12   order by color")
13    (brl ]<li><strong>
14  <a href="p2.brl?[
15  (brl-url-args brl-blank? color)
16  ]">[(brl-html-escape color)]</a></strong>
17  ]))]</ul>
18
19  <p>Count: [rowcount]</p>
20
21  </body>
22  </html>
```

We see in line 16 that the color name is properly escaped before being used as the text of an anchor.

It would be a waste to use a 22-line example just to illustrate one procedure, so the `brl-url-args` syntax is also illustrated in line 15. The first argument to `brl-url-args` should generally be `brl-blank?`. If you want to draw a distinction between inputs that have been sent as blank strings and inputs that have not been sent at all, you might use `null?` instead, but this is generally a bad idea. The remaining arguments in the `brl-url-args` syntax should be variable names. The variable names and their values are encoded in such a way as to be included in a URL. In the example above, a URL is generated that, when followed, goes to a page '`p2.brl`' and `color` is given as an input.

The `brl-url-args` syntax is convenient in that it omits blank inputs, resulting in nice, compact URLs. But sometimes you may want to simply URL-escape a value. In that case, use the `brl-url-escape` procedure exactly as `brl-html-escape` is used.

4.2.6 Searches and Sortable Columns

The next example, '`colors.brl`', is intended to be the action of an HTML form to search the database by name, color or both. The results appear in columns which can be sorted by clicking[1] on a column header. Successive clicks on a header reverse direction. If you're looking at the list in say, reverse order by name and then click on the color header, the list will then be sorted by color, but within each color they will be sorted in reverse order by name.

```
 1 <html>
 2 <head>
 3 <!-- [
 4  (inputs color name order1 order2)
 5  (define where-clause
 6    (string-append "where 1=1"
 7      (if (brl-nonblank? name)
 8          (string-append " and name=" (ss name))
 9          "")
10      (if (brl-nonblank? color)
11          (string-append " and color=" (ss color))
12          "")))
13  (define (valid-order str)
14    (if (or (brl-blank? str)
15            (sql-order-member str '("name" "color")))
16        str
17        ""))
18  (set! order1 (valid-order order1))
19  (set! order2 (valid-order order2))
20  (define nonblank-orders (brl-nonblanks (list order1 order2)))
21  (if (null? nonblank-orders) (set! nonblank-orders '("name")))
22  (define order-columns (brl-string-join ", " nonblank-orders))
23  (define (sort colname)
24    (string-append
25      "colors.brl?"
26      (brl-string-join brl-url-arg-separator
27        (brl-nonblanks
28          (list
29            (brl-url-args brl-blank? name color)
30            (brl-url-arg-seq "order"
31              (sql-order-prepend colname nonblank-orders) 1))))))
32  (define conn (db1 brl-context))
33  (define st (brl-sql-statement brl-context conn))
34 ] -->
35 <title>Search Results</title>
36 </head>
37
38 <body>
39
```

[1] more precisely, activating the link

```
40 <table>
41  <tr>
42    <th><a title="sort" href="[(sort "name")]">Name</a></th>
43    <th><a title="sort" href="[(sort "color")]">Color</a></th>
44  </tr>
45 [(define rowcount
46    (sql-repeat st (name color)
47 ()select name, color
48 from favcolor
49 [where-clause]
50 order by [order-columns)
51      (brl ] <tr>
52    <td>[(brl-html-escape name)]</td>
53    <td>[(brl-html-escape color)]</td>
54  </tr>
55 ()))]
56 </table>
57
58 <p>Found: [rowcount] [where-clause]</p>
59 </body>
60 </html>
```

Normally a search query has conditions that are always part of the where clause. In this example there are none except the search conditions, but we've put `1=1` in as a placeholder (line 6). Lines 7-9 specify a name if one was typed in the appropriate input. Lines 10-12 do the same for color. If there were other search criteria, we could add them in much the same way.

The order1 and order2 inputs are URL arguments that '`colors.brl`' sends to itself in order to do column sorting.

Putting arbitrary strings from form input into your SQL query is a risky proposition: there's no guarantee that the `order1` and `order2` inputs will be what you intended. Lines 13-17 define a function that returns a valid value for either of these variables regardless of its input. This function is used in lines 18 and 19. Line 20 gets a list of all the non-blank order columns. This is the last line that would have to change if you were to add another column.

Line 21 makes the default sort be by name. Line 22 gets a comma-separated list of columns to sort by. Lines 23-31 define a function that returns a relative URL that points back to '`colors.brl`' with arguments for the search parameters and the sort columns. The new BRL functions introduced here are `brl-url-arg-seq`, which returns a string of URL parameters with sequentially-ordered names, and `sql-order-prepend`, which takes a column and a list, and returns a new list with that column at the front. If the column was previously at the front, it is toggled between ascending/descending order.

Lines 32-41 are old material. Lines 42 and 43 show how the sort function is used (defined in lines 23-31). Lines 47-50 show a dynamically-generated SQL statement that incorporates the search criteria and sort columns.

4.3 String Manipulation

Scheme has basic string functions which are described in R5RS and work in all Scheme implementations. The Kawa implementation of Scheme has additional string functions described in the Kawa manual. BRL provides yet more string functions, the most important of which are described here.

Any object can be converted to a string with the `brl-string` procedure. But there are more efficient and more powerful means of converting objects to strings if you know something about the object.

4.3.1 Numbers to Strings

Scheme (and thus BRL) includes a procedure called `number->string` whose only formatting capability is an optional argument specifying a number base. E.g. `(number->string 255)` yields `"255"`, and `(number->string 255 16)` yields `"ff"`.

For other formatting capabilities, the `brl-format` procedure provides a simple interface:

- `(brl-format 65535 "#,###")` yields `"65,535"`
- `(brl-format 1/3 "$0.00")` yields `"$0.33"`
- `(brl-format 1 "#%")` yields `"100%"`

For more details, see your Java documentation for `java.text.DecimalFormat`. If BRL is later ported to a non-Java platform, `brl-format` will still work the same way.

If you will be converting a lot of numbers to strings of the same format, `brl-decimal-formatter` will be efficient and convenient.

```
> (define money (brl-decimal-formatter "$#,##0.00" "NULL"))
> (money (* 100 100))
$10,000.00
> (money 1/3)
$0.33
> (money (list))
NULL
```

As can be seen in the above example, `brl-decimal-formatter` takes two arguments, the first being a string suitable for use with `brl-format`, and the optional second argument being a string to use for null values. The return value of `brl-decimal-formatter` is a procedure that takes one argument and returns a string. If the concept of a procedure that returns a procedure seems confusing, study the example above for a minute. You'll see it's actually quite simple.

4.3.2 Dates to Strings

In addition to numbers, `brl-format` can also do simple date formatting, e.g. `(brl-format (brl-now) "yyyy-MM-dd")` might yield `"2000-05-19"`.

For more details, see your Java documentation for `java.text.SimpleDateFormat`. If BRL is later ported to a non-Java platform, `brl-format` will still work the same way.

Your site should have a standard format for dates that is used everywhere. It is best to define such a format in your '`sitedefs.scm`' file using the `brl-simple-date-formatter` procedure.

```
(define date10 (brl-simple-date-formatter "yyyy-MM-dd" "NULL"))
```

You would then use the `date10` procedure anywhere a date should be formatted into a string of at most 10 characters.

4.3.3 Trimming, Splitting and Joining

To trim the whitespace off of both sides of a string, use the return value of `(brl-trim str)`, which returns a trimmed version. It does not modify the string passed to it. If you want to modify a string, do this: `(set! str (brl-trim str))`

Use `brl-split` to split a single string into a list of strings based on a separator string, e.g. `(brl-split " and " "lions and tigers and bears")` yields `("lions" "tigers" "bears")`.

To join the string representations of a number of objects into a single string with a separator, use `brl-string-join`. For example, `(brl-string-join ", " (list "one" 'two (+ 2 1)))` yields `"one, two, 3"`.

4.4 Sending e-mail

Here is example code for '`sitedefs.scm`' to set things up to send mail via an SMTP server:

```
(define mail
  (brl-smtp-sender
   "mail.example.com"           ; the SMTP server
   "www.example.com"            ; the web server
   "webmaster@example.com"      ; bounced mail goes here
   ))
```

This creates a procedure called `mail` that takes at least two arguments. The first argument is a list of e-mail addresses that are recipients of the message. The second and subsequent argument become a string that forms an e-mail message, i.e. Internet mail headers followed by a blank line, then the body of the message.

The following example would be the ACTION page for an HTML form:

```
 1  <html>
 2  [(inputs email yourname quest colour)
 3   (define msg
 4     (string-append
 5  ]From: [email]
 6  To: strangeman@example.com
 7  Subject: questions three
 8
 9  What is your name?              [yourname]
10  What is your quest?             [quest]
11  What is your favourite colour?  [colour]
12  ])
13   (mail '("strangeman@example.com") msg)
14  ]
15  <body>
16
```

```
17  <p>The following e-mail message was sent:</p>
18
19  <PRE>[(brl-html-escape msg)]</PRE>
20
21  </body>
22  </html>
```

Line 2 defines the inputs that come from the HTML form. Line 3 defines a variable called `msg` to be simply the concatenation of several strings, which we find in lines 5-12. Then on line 12 we have close parens for `string-append` and `define`. Finally, on line 13, the mail is sent using the `mail` procedure that was defined in '`sitedefs.scm`' previously.

4.5 Connection Pools and Other Java Objects

The Kawa manual describes `make`, `invoke`, and `invoke-static` syntax for manipulating Java objects. This section will illustrate their use within '`sitedefs.scm`'. You should define procedures that manipulate Java objects only in this file, then use those procedures in individual pages. This methodology will keep the individual pages clean and portable.

The following example uses DbConnectionBroker, but could easily be adapted to use another connection-pool object.

```
1  (define testdb-pool
2    (make <com.javaexchange.dbConnectionBroker.DbConnectionBroker>
3      "com.internetcds.jdbc.tds.Driver"
4      "jdbc:url-here"
5      "user-here" "password-here" 1 5 "/tmp/testdb.log" 1))
6
7  (define (db1 brl-context)
8    (let ((c (invoke testdb-pool 'getConnection)))
9      (brl-prepend-endproc!
10      brl-context
11      (lambda () (invoke testdb-pool 'freeConnection c)))
12     c))
```

When the BRL servlet starts or re-reads '`sitedefs.scm`', a variable `testdb-pool` (line 1) is bound to a new object (2). In this example, an 8-argument constructor is used. Then, the `db1` procedure is defined to get a connection from this pool (line 8), then prepend to the list of things to do when a BRL-page request is complete (line 9), a procedure that takes no arguments and frees the connection (line 11). The return value is the connection (line 12).

Note that no changes are required for individual pages to take advantage of the connection pool. Previously, `db1` was a procedure that opened a connection and assured that it was closed later. Now it is a procedure that gets a connection from the pool and assures that it is freed later. The individual pages don't know the difference.

5 BRL Reference

5.1 String Functions

`brl-string`
: takes an arbitrary number of arguments of any type, and returns a string that is the concatenation of their `display` representations.

`brl-simple-date-formatter`
: takes a string argument that specifies a format according to `java.text.SimpleDateFormat` specifications, an optional second string argument to use for null values, and returns a procedure that takes a date or null argument and returns an appropriately formatted string.

`brl-decimal-formatter`
: takes a string argument that specifies a format according to `java.text.DecimalFormat` specifications, an optional second string argument to use for null values, and returns a procedure that takes a real or null argument and returns an appropriately formatted string.

`brl-format`
: takes a number or date argument followed by a string argument specifying an appropriate format, and returns a string representation of the first argument.

`brl-string-escaper`
: takes an argument that is a list of pairs, the car of each pair being a character and the cdr being a string that represents the escape sequence for that character. It returns a procedure that takes one string argument and returns that string if no escaping is needed, or an escaped version otherwise.

`brl-html-escape`
: takes a string argument and returns a string with the following characters appropriately escaped for HTML: `< > " &`

`brl-scheme-escape`
: takes a string argument and returns a string with backslashes and double quote characters appropriately escaped for use in Scheme strings. Usually Scheme's `write` procedure is better for this purpose.

`brl-msft-escape`
: takes a string argument and returns a string with certain non-standard characters converted into ASCII-standard single and double quotes.

`brl-sql-escape`
: takes a string argument and returns a string with single quote characters appropriately escaped for use in Scheme strings. Usually `brl-sql-string` is better for this purpose.

`brl-sql-string`
: takes a string or null argument and returns either a SQL string enclosed in single quotes and properly escaped, or the string `"NULL"`.

`brl-sql-number`
: takes a number, string or null argument and returns a string representation of the number or `"NULL"` as appropriate. An error will be thrown if a string argument does not represent a number.

`brl-latex-escape`
: takes a string argument and returns a string with backslash-escaping for characters that are special to the LaTeX typesetting language.

`brl-ends-with?`
: takes a suffix argument and a string argument, and returns false unless the string ends with the suffix.

`brl-starts-with?`
: takes a prefix argument and a string argument, and returns false unless the string starts with the prefix.

`brl-nonblank?`
: takes any argument, and returns false only if the argument is null, the empty string, or false.

`brl-blank?`
: takes any argument, and returns false unless the argument is null, the empty string, or false.

`brl-nonblanks`
: takes a list argument, and returns a subset of that list containing only non-blank items.

`brl-trim` takes a string argument and returns a string formed by trimming whitespace from both sides.

`brl-split`
: takes a string argument specifying a separator, then a string argument to split, and returns a list of strings found in the second argument delimited by the first argument.

`brl-string-join`
: takes a string argument specifying a separator, then a list argument of objects. A string is returned that concatenates the `display` representations, separated by the separator string. This function is named in a different style from `brl-trim` and `brl-split` to avoid confusion with SQL joins.

5.2 Web Functions

`brl-referer-ok?`
: takes a BRL context argument and returns false only if the HTTP Referer header has been supplied and indicates that the referring page is on a different site from the current page.

`brl-referer-check`
: takes a BRL context argument and throws an error if `brl-referer-ok?` returns false, otherwise an empty string is returned.

`brl-http-status!`
: takes a BRL context argument, an HTTP status code and a string. The HTTP status for the current page is set.

`brl-context-cookies`
: takes a BRL context argument and returns a list of lists. Each inner list consists of a cookie name (symbol), a cookie value (string), and possibly other elements.

`brl-cookie-value`
: takes a BRL context argument and a symbol argument representing a cookie name. A string representing the value of that cookie is returned.

`brl-cookie-set!`
: takes a BRL context argument, a symbol argument representing a cookie name, a string argument representing the value of that cookie, and any optional arguments. Optional arguments must be in pairs that change attributes of the cookie: `comment:` *string*, `domain:` *string*, `maxage:` *int* (seconds), `path:` *string*, `secure:` *boolean*, `version:` *int*.

`brl-url-arg`
: takes three arguments, the first being a procedure, usually `brl-blank?` but sometimes `null?`, which is applied to the third argument. If the result is false, a URL argument string of the form *name=value* is returned, using the second argument as the name and the third argument as the value. Otherwise an empty string is returned.

`brl-url-args`
: This syntax takes two or more arguments. The first argument is a procedure as with `brl-url-arg`. Second and subsequent arguments should be variable names. A string is returned suitable for inclusion in a URL with the variable names used as parameter names and the variable values used as parameter values.

`brl-url-contents`
: takes a string argument (URL), fetches its contents and returns the results as a string.

`brl-html-options`
: takes two arguments, the first being a "selected" value, the second being a list of lists. For each inner list, the first element is a value for an HTML `OPTION`

tag, and the second element is a string with which to label the option. A string is returned with appropriate HTML `OPTION` tags.

`brl-get-update`
: This syntax takes two or more arguments. The first argument is a `brl-context` structure that includes input from a web form. The second and subsequent arguments must be of the form (*validator varname*), where *validator* is a procedure for validating an input value, and *varname* is an input of the form. The return value is a list of lists. For each inner list, the first element is the name of an input that has changed from its old value (i.e. the value of an input whose name is formed by prepending "o" to name of the input in question), and the second element is the new value.

5.3 SQL Functions

`sql-driver`
: takes a string argument. For the JVM-based implementation, this string represents the class of a JDBC driver. Future implementations may require different strings. The return value is implementation-specific.

`sql-connection`
: takes a string argument representing a database to connect to, and optional second and third string arguments representing a username and password. For the JVM-based implementation, the first argument is a JDBC URL, and `sql-connection` may also take exactly two arguments, the second of which is a `<java.util.Properties>` object. An SQL connection object is returned.

`sql-connection?`
: takes any object as its argument, and returns false unless that object is an SQL connection object.

`sql-connection-close`
: takes an SQL connection argument and closes it.

`brl-sql-connection`
: takes two or more arguments, the first of which is a `brl-context` structure representing a request. Subsequent arguments are passed to `sql-connection`. The resulting SQL connection is returned, and will be closed when the request is exited for any reason.

`sql-statement`
: takes an SQL connection argument and returns an SQL statement object that can be used for queries and updates.

`sql-statement?`
: takes any object as its argument, and returns false unless that object is an SQL statement object.

`sql-statement-close`
: takes an SQL statement argument and closes it.

`brl-sql-statement`
: takes two arguments, the first of which is a `brl-context` structure representing a request. The second argument is passed to `sql-statement`. The resulting SQL statement is returned, and will be closed when the request is exited for any reason.

`sql-execute`
: takes a statement argument. Subsequent args are passed to `brl-string` to form an arbitrary SQL statement. Useful for table definitions, etc.

`sql-execute-update`
: takes a statement argument. Subsequent args are passed to `brl-string` to form an SQL insert, update or delete. The number of rows affected is returned.

`sql-statement-results`
: takes a statement argument and a string argument representing an SQL query. Query result values are returned in the form of a list of lists.

`sql-execute-query`
: takes a statement argument and a string argument representing an SQL query. Query results are returned in the form of an SQL result-set object object that includes meta-data and the potential to retrieve values.

`sql-resultset?`
: takes any object as its argument, and returns false unless that object is an SQL result-set object.

`sql-rsmd` takes an SQL result-set argument and returns an SQL result-set meta-data object.

`sql-rsmd?`
: takes any object as its argument, and returns false unless that object is an SQL result-set meta-data object.

`sql-rsmd-column-labels`
: takes an SQL result-set meta-data argument and returns a list of column labels.

`sql-rsmd-column-names`
: takes an SQL result-set meta-data argument and returns a list of column names.

`sql-rsmd-column-typenames`
: takes an SQL result-set meta-data argument and returns a list of column type names, which are implementation-specific.

`sql-resultset-nextrow`
: takes an SQL result-set argument and an integer argument representing the number of columns to retrieve. A list of result values is returned for the next row from the SQL query.

`sql-repeat-rsmd`
: This syntax takes five or more arguments. The first argument is a variable name which will be bound to an SQL result-set meta-data object within the

scope of the `sql-repeat-rsmd` expression. The second argument is an open SQL statement object. The third argument is a formal arguments list legal for use with `lambda`. The fourth argument is of the form `)`*obj ...*`)`, consisting of objects, usually strings, which are used to build a query string. Fifth and subsequent arguments are expressions which form the body of a procedure with arguments as specified by the third argument. This procedure is repeatedly applied to result-set rows, and the number of rows is returned.

`sql-repeat`
: This syntax takes four or more arguments exactly like the second and subsequent arguments to `sql-repeat-rsmd`, but no provision is made to capture the result-set meta-data.

`sql-order-prepend`
: takes a string argument and a list argument. The first argument and the elements of the second argument are strings that represent columns to order SQL results by, and may or may not be followed by "desc" to indicate descending order. A list is returned whose first element is the first argument, reversed by removing or adding "desc" as appropriate if that column was already the first element of the list. Otherwise the first occurrence later in the list is deleted.

`sql-order-eqv-di?`
: takes two string arguments and returns false only if they differ other than by the suffix `" desc"`. Case is ignored.

`sql-order-desc?`
: takes a string argument and returns false unless the string ends in `" desc"`. Case is ignored.

`sql-order-member`
: takes a string argument and a list argument, and returns false only if there is no member of the list that is equivalent (ignoring the suffix `" desc"`). than by the suffix `" desc"`. Case is ignored.

`sql-order-reverse`
: takes a string argument and returns a copy of the string with `" desc"` appended if it wasn't already there, deleted otherwise.

`sql-partial-update`
: takes three or more arguments. The first argument is an SQL statement object. The second is a string, the name of a table to update. The third is a list like that returned by `brl-get-update`. An SQL update is performed if appropriate, with the fourth and subsequent arguments appended to the string forming the query.

5.4 Miscellaneous Functions

`brl-implementation-version`
: takes no arguments and returns a string corresponding to the BRL release number.

`brl-readall`
: takes an input port as an argument and returns a list of Scheme objects read from that port using the BRL template syntax.

`brl-hash`
: takes no arguments and returns a hash table.

`brl-hash?`
: takes any Scheme object as an argument and returns false unless that object is a hash table.

`brl-hash-size`
: takes a hash table as an argument and returns the number of items stored therein.

`brl-hash-put!`
: takes a hash table argument, a key argument and a value argument. The key/value pair is stored in the hash table.

`brl-hash-get`
: takes a hash table argument and a key argument, and returns the value if any exists in the hash table for that key, otherwise returns false.

`brl-hash-remove!`
: takes a hash table argument and a key argument, and removes the key/value pair from the hash table.

`brl-hash-keys`
: takes a hash table argument and returns a list of all keys for that hash table.

`brl-hash-contains-key?`
: takes a hash table argument and a key argument, and returns false unless the hash table contains that key.

`brl-random`
: takes one argument. If that argument is a positive exact integer N, a pseudo-random number M is returned such that 0 <= M < N. If that argument is an input port, a few bytes are read from the port to help seed the random number generator. The same input is not guaranteed to produce the same random number sequence each time.

`brl-random-typeable`
: takes a positive integer argument N and returns a string of length N consisting of random numbers and lowercase letters, excluding 0, 1, l, o.

Other functions exist and will be documented later. The curious can peruse '`gnu/brl/*.scm`' in the source distribution.

Appendix A The Whys of BRL

A.1 Why not HTML-like syntax?

Many web template engines enclose program code in something that looks like an HTML tag. CFML takes this to an extreme by making most of its language constructs look like HTML tags. This may make for some comfort for someone used to HTML, but that comfort goes away as one recognizes the fundamental cognitive difference between markup and programming code. Writing a web page and programming a computer to write a web page are two different mental activities, and a different syntax helps one switch back and forth between them.

Another argument for HTML-like syntax is the hope of having valid SGML pages for the template source code. I have yet to see an actual SGML or XML tool for manipulating such source code. More important than valid source code is valid output. A valid source file might produce invalid output or vice versa.

HTML-like code, e.g. <%=name%> likes <%=color%>, if brought up as source code in a browser, shows up only as the word "likes". The BRL code [name] likes [color] shows up completely, and is more useful for assessing what the output might look like. Square brackets are used just as they are in English.

string[?, Why not HTML-like syntax?, The Whys of BRL,]

A.2 Why Scheme?

BRL is a syntactic descendant of cgiemail, a program written in 1995 to allow non-programmers to specify the exact format of e-mail messages sent from HTML forms. Its template language began very simply. One would write the literal text of an e-mail message, putting [inputname] wherever an HTML input's value should go. Over time, more functionality was added. One could name an input with a required- prefix to indicate that an error should be signaled if that input was left blank. One could use [$VAR] to put in a CGI environment variable. Text could fill a given number of columns with C-like [%-9.9s,inputname].

As more functionality got added and more special characters were used, I could see my simple template system potentially evolving into a programming language that looked like line noise. A function should be spelled out rather than having its own special character. I wanted to preserve the simple [inputname] syntax, and just have one special character that indicated a function name rather than a variable name would follow. I also wanted to have a means of combining functions.

So, perusing my keyboard for that one special character I happened on the parens and remembered Scheme from a Computer Science course 10 years earlier. That syntax turned out to be a perfect fit. Scheme's syntax looks a lot like an imperative sentence in English, but with less ambiguous grouping: (verb object1 object2 ...). And as a bonus, the simple [inputname] syntax would also work, as a variable name all by itself is a valid Scheme expression.

Scheme's simple syntax for defining procedures is also very helpful in web application development. Rapid prototyping is possible by creating little procedures as needed within

an individual page. It is then trivial to move from quick prototype to MVC separation,[1] often just by cutting code from an individual page and pasting it into '`sitedefs.scm`'.

As a contrasting example, Java has such a large overhead for moving code out of an individual JSP page and into a Java bean that it is unlikely that a programmer will spend the effort. This problem has motivated the creation of systems that force MVC separation from the start (e.g. webmacro), making rapid prototyping more difficult. Thanks to Scheme, BRL does not have this RAD vs MVC dilemma.

Some programmers dislike Scheme because mathematical expressions tend to look very different from how they would normally be written. This is of little importance to BRL because math is rarely used in web pages, and when it is used it tends to be simple.

Some programmers think they will have a hard time switching from the verb(object1, object2, ...); syntax that they use in other popular languages. However, a good programmer doesn't usually take long to adjust to an unfamiliar syntax. It may actually be less confusing to switch back and forth between dissimilar syntaxes than to switch back and forth between syntaxes that are similar but not identical, e.g. between PHP and Perl.

A.3 Why]string[?

Even for someone accustomed to Scheme, one aspect of BRL's syntax is confusing. An expression like (string-length]string[) looks like mismatched parentheses. Yet there is good reason to allow this syntax.

In other compiled template systems like JSP, literal text outside of delimiters is converted into print statements. These are combined with the statements inside the delimiters to produce source code that is compiled. This allows for interesting uses of the template system, e.g.

```
<% for (i=0; i<10; i++)
   {
%>  <li> <%=name[i]%> likes <%=color[i]%>
<% } %>
```

The concise template syntax is used with flow constructs such as loops and conditionals. Note that the `<% } %>` looks mismatched, but makes sense when you know how things work. It's also nice for a non-programmer looking through the code and simply scanning for `%>` to see where the programming stops and the HTML starts. In BRL 1, something like this JSP example was not possible. You could conditionalize around regular Scheme syntax, but not around syntax that included the `[` and `]` delimiters.

BRL 2 (described in this manual), thanks to Scheme, is able to take this concept one step further.

Its concise template syntax can be used not only with flow constructs, but within any language construct. Unlike any other template language, BRL lets the same syntax to be used to construct not only the HTML page to be output, but also e-mail messages, complex SQL queries, or anything else that mixes static and dynamic content.

For this reason, one cannot treat BRL as Yet-Another-Language Server Pages. One's thinking has to be adjusted. As a starting point, one might look at the JSP example above

[1] Model-View-Controller separation is a concept more applicable to stand-alone graphical interfaces than server-side web applications, but is used here for lack of a better term.

and imagine `[` and `]` in place of `<%` and `%>`, and `(brl` and `)` in place of `{` and `}`. Then spend some time looking at the e-mail example in this manual (see Section 4.4 [Sending e-mail], page 19). A relatively small investment of time acclimating to BRL syntax will repay handsomely in a powerfully expressive tool for writing dynamic web pages.

Index

Table of Contents