BRL

A database-oriented language to embed in HTML and other markup
Release 2.2, 6 June 2003

Bruce R. Lewis
Eaton Vance Management

1 Introduction

1.1 What is BRL and Why Should I Care?

BRL stands for *Beautiful Report Language*.

- *Beautiful*: It is easy to write BRL code that is understandable and maintainable, appealing to a programmer's sense of aesthetics.
- *Report*: BRL is particularly suitable for constructing output that is a mix of static and dynamic content, e.g. web pages, e-mail messages. Its greatest strength is constructing output from SQL databases.
- *Language*: The full power of a general-purpose programming language is there when needed. Simple examples are trivial uses of the language, but look more like templates than programming. The template system and programming language are more tightly integrated in BRL than in any other system.

Given this glowing description, a fair question is, why aren't more people using BRL instead of the 20 or so more prevalent tools out there for server-side web programming? There are several answers.

People haven't heard of BRL. There is no aggressive marketing campaign for it, other than this manual. Few people actually read this manual, because they think all server-side web tools are essentially the same. Why look at one more? Examples in this manual will show how BRL is different.

People use a win-stay, lose-shift strategy in their choice of software tools. Once they are familiar with one tool's strengths and weaknesses, they avoid the unknown. They develop strategies to work around systematic problems with the tool. Only when they see someone else doing better work in less time will they get the perception that they're "losing" and consider "shifting" to a different tool.

People hold misconceptions about Scheme. BRL is a "gentle-slope" dialect of the Scheme programming language. Scheme is taught in hundreds of universities, colleges and secondary schools worldwide. Scheme uses a syntax that is much less complicated than the C/Java/Perl family of syntaxes. 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.

Here are answers to the most common misconceptions:

1. *Scheme is a difficult language.* This is true only when it is used to solve difficult problems. For simple tasks, its simplicity is hard to beat. Students who use Scheme in a CS course that tackles difficult problems often leave thinking it's the language that's difficult.
2. *Scheme is slow.* This may have been true in 1986, but isn't true today. In a simple benchmark, BRL outperformed PHP3, a tool used in many successful web sites. Even more performance could have been gained by choosing a faster servlet engine (JServ was used).
3. *Scheme lacks an extensive function library.* It's true that the standard Scheme specification is small. However, individual implementations invariably include more, including

a "foreign function interface" allowing use of C or Java libraries. See Section 4.10 [Connection Pools and Other Java Objects], page 32, for an example of using Java objects while keeping individual BRL pages "pure."

SLIB

4. *Since Scheme is not prevalent in industry, it will be hard to find programmers.* Programmers need to learn new tools throughout their careers. Those involved with server-side Java in particular must expose themselves to a constantly-changing variety of tools designed to keep Java code out of web pages. The time needed to learn enough Scheme for BRL programming is only a small addition to the time needed to get up to speed on a new project. A programmer with a Computer Science degree has likely done Scheme programming already. College is less likely to expose a student to WebMacro, Velocity or Tea.

1.2 About This Manual

This manual must cater to two audiences. One audience is investigating whether BRL is worth installing and trying out, and is interested in comparing BRL with competing tools. Another audience has already decided to use BRL and just wants to know how.

See Chapter 4 [Learning BRL by Example], page 13, if you're casually investigating BRL for comparison with other tools. If you get lost, skip back to the Intoducing Scheme chapter.

If you want to learn how to program with BRL, take this manual sequentially. While reading the BRL by Example chapter and the BRL Reference chapter, stop frequently to try out what you've learned.

In either case, please do give feedback as to how well the manual is working for you, or about additional features you would like to see in BRL. E-mail is appreciated. To: `brlewis@users.sourceforge.net`

Advocacy: This label at the beginning of a paragraph indicates material that is only useful for comparing BRL to other tools. If you've already decided on BRL and are just trying to learn how to use it, skip ahead to the next section break.

1.2.1 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?

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

Installing BRL on your own web server involves installing and configuring a Java Web Application or Servlet. Most users will be able to install BRL using only the first section of this chapter. The remaining sections are needed only when using the obsolete jserv.

2.1 Easy Installation via learnbrl.war

Configuration of servlet containers has gotten a lot easier since this chapter was first written. A standard has been established for web applications such that they can be packaged up in a .war file that includes all the necessary .jar files and configuration directives.

The learnbrl.war file is one such web application. See Section 2.3 [Downloading BRL], page 4, but note that you may have to scroll down past the brl and kawa-brl packages to the bottom of the download page to find learnbrl.

For best use of the learnbrl tutorial, you should create a '`learnbrl`' directory under your servlet container's '`webapps`' directory. There you can unpack learnbrl with the following command:

```
jar xvf learnbrl.war
```

If you use a shared hosting service, use the above command in the directory that contains a WEB-INF subdirectory. You will need to edit WEB-INF/web.xml to change the parameter `cut-uri` servlet parameter to `false`. Follow your service provider's instructions for restarting your servlet engine.

Most shared hosting services use Apache. A convenient way to get started is to configure Apache to run .brl files through BRL. In the long run, it's better not to name URLs by the technology behind them, which might change over time. As a learning convenience, put the following in a '`.htaccess`' file in the directory where you'll put BRL files:

```
Action brlsv /servlet/brlsv
AddHandler brlsv brl
```

This will facilitate you modifying the example files that are part of the tutorial.

2.2 Prerequisites

If you're using a Java web application server like Paperclips, Jetty, or Tomcat (among others), then everything you need to know was in the previous section. If you're using the obsolete JServ, this section and the ones that follow will be useful.

Before you can run BRL, you need the following:

1. A servlet engine that supports the Servlet API 2.1 or later.
2. Release 1.7 of Kawa, available from the BRL project page and elsewhere. See Section 2.3 [Downloading BRL], page 4. BRL will likely also work with future Kawa releases as they come out, but it may be necessary to recompile.

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 (potato release), 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>
```

2.3 Downloading BRL

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

2.4 Precompiled JAR Installation

1. Put the compiled jar files, ‘brl-2.2.jar’ and ‘kawa-1.7.jar’ into the CLASSPATH of your servlet engine. For example, if you are using JServ, add lines like the following to ‘jserv.properties’:

   ```
   wrapper.classpath=/usr/local/share/java/brl-2.2.jar
   wrapper.classpath=/usr/local/share/java/kawa-1.7.jar
   ```

2. Extract ‘brl-global.scm’ and ‘sitedefs.scm’ from the BRL jar 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’.

   ```
   jar xvf brl-2.2.jar gnu/brl/brl-global.scm gnu/brl/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 jar 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. **Note:** It has been reported that WEB-INF will not work with the Tomcat servlet engine, since the BRL servlet cannot access this directory after it has initialized. It needs to check if ‘sitedefs.scm’ has been modified and reload it when necessary.

   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.

2.5 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 jar file to put in your class path:

```
make all brl.jar
```

Note that the resultant '`.jar`' file has a name that includes the current version number, e.g. '`brl-2.2.jar`'. Follow the "Precompiled JAR 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.

2.6 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.

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.

If a different error occurs, then something unusual about your servlet engine is causing problems. See Section 1.2.1 [Further Help], page 2.

3 Introducing Scheme

It's fairly easy to pick up basic BRL syntax just by looking at examples. However, learning some Scheme beforehand can clarify the subtleties of examples in this manual and help you go beyond the examples on your own.

The Scheme programming language was designed with teaching in mind. There is a very friendly tool available for learning it called DrScheme.

Alternatively, 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 nearly 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 your Scheme implementation. But the result is the same. You type `"Hello, World"` (*including* the quotes) and you get `"Hello, World"`. This may be surprising if you're used to other computer languages in which `"Hello World"` can only be used as an argument to a command. Scheme does not have this command/argument dichotomy. You use the term *expression* for both concepts. 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.

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

This illustrates the most common way to combine expressions. The expression `"!!"` is a constant string, much like the `"Hello, world"` you typed earlier. To the left of that is `h`, the variable we just bound to a string value. Then there's `string-append`, a function that concatenates strings.

An important thing to note here is that `string-append` is just another variable, like `h`. The only difference is that `h` is bound to a string value, while `string-append` is bound to a *procedure* value. Procedures are just another Scheme data type, like strings or numbers.

This makes the process of evaluating `(string-append h "!!")` simple and uniform. Each expression within the parentheses is evaluated, then the value of the first expression (in this case `string-append`) is *applied*, i.e. invoked with the remaining values as function arguments.

Caution: If you're accustomed to other languages in which adding extra parentheses does not change the meaning, you'll need to unlearn that habit with Scheme. For example `h` is just the variable h, but `(h)` means to treat `h` as a function with no arguments and apply it.

Let's review how this expression is evaluated:

```
(string-append h "!!")
```

First, each sub-expression within the parens is evaluated.

1. `"!!"` is self-evaluating; its value is `"!!"`.
2. `h` is a variable; the value bound to it is `"Hello, World"`.
3. `string-append` is a variable; the value bound to it is a procedure.

This makes the expression equivalent to

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

Next, the procedure bound to `string-append` is applied to the arguments `"Hello, World"` and `"!!"`, returning the value `"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*, *bound*, 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. Although simple and uniform, this syntax can be a little confusing at first for math:

```
(+ (* 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 `(string? name)` evaluates true[1], then the second argument `(greet name)` is evaluated. Otherwise the third argument `(greet "NON-STRING!")` 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.

Note that `number->string` is just a regular function name. There is nothing special about the characters `->` in the middle of a Scheme identifier.

3.3 Data Structures

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.

[1] Any Scheme value other than `#f` is true, not just `#t`.

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

4.1 Web Page for Bruce

The example below, '`bruce.brl`', illustrates the basics of BRL.

```
[(define myname "Bruce")
 (define my "Bruce's")]

<pre>
This is [my] web page.
[myname myname
 myname myname]

[my] favorite person is [[your name here].
[my] favorite number is [(brl-random 2)].
</pre>
```

When you visit this page, the web server uses BRL to transform this page into regular HTML to be sent back to your browser. Your browser does not need to know anything about BRL to view this page. Here is the output sent to your browser:

```
<pre>
This is Bruce's web page.
BruceBruceBruceBruce

Bruce's favorite person is [your name here].
Bruce's favorite number is 0.
</pre>
```

As you can see, the `define` expressions do not result in any output. They merely define variables (`myname` and `my`) for later output. Variable substitution transforms "This is [my] web page" into "This is Bruce's web page."

The "BruceBruceBruceBruce" output shows that spaces and line breaks do nothing *inside* square brackets. Values of multiple expressions are output with no space in between. *Outside* of square brackets, spaces and line breaks are sent verbatim.

An important note on the "[my] favorite person is [[your name here]" line is that outside square brackets there is only *one* special character, the open square bracket. If you want an open square bracket to appear in the output, it needs to be "escaped" by putting two. The close square bracket is not special and does not need to be escaped.

The `(brl-random 2)` expression outputs 0 or 1, so different visitors will see different results.

Advocacy: Syntax for variable substitution is extremely intuitive. Square brackets are used just as in English, i.e. text outside the brackets is a literal quotation; text inside the brackets is not.

The first thing that speeds BRL programming is that square brackets are easier to type than the combination "angle-bracket plus something else" syntax used in ASP, JSP and PHP. Such syntax is billed as more standards-compliant, but generally isn't. It's often used in entity attributes, e.g. `<img src="<%=myurl%>">`, where angle brackets are illegal. Other

illegal angle brackets, e.g. the less-than operator, show up in embedded code. Even if that syntax is used in such a way as to produce a standards-compliant source file, there is no guarantee that standards-compliant output will be produced.

Static HTML content tends to have a lot of angle brackets. Delimiting dynamic content with angle brackets makes it difficult to visually pick out. BRL's square-bracket syntax makes it easy, without the need for a syntax-highlighting editor.

Non-literal expressions other than variable substitution are quickly spotted by the opening paren, e.g. [(brl-random 2)] is quickly recognized as more than variable substitution. A more common syntax for programming languages would be [brl-random(2)], which would take slightly longer to recognize as more than variable substitution. Small improvements in readability add up to a more beautiful language.

4.1.1 Understanding BRL Inside-Out

This next example will go one step farther with the `(define myname "Bruce")` expression in Web Page for Bruce.

```
[(define myface ]
<img src="/img/myface.jpg" alt="Bruce's face">
[)]
```

Prior to now, it looked as if BRL was just Scheme expressions enclosed in square brackets. This example breaks that mental model. It looks like a `define` expression that starts within one set of square brackets ends within another set. However, there's another way of looking at it.

BRL's use of square brackets is the opposite of what you might think. It isn't that [marks the start of Scheme code and] marks the end. Instead,] marks *the start of a string* and [marks *the end of a string*. For example, `(define myname "Bruce")` could also be written as `(define myname ]Bruce[)`. The example above differs only in the amount of text between] and [.

Additionally, strings can be delimited by the beginning and end of a BRL file.

The best way to think of BRL is as a Scheme interpreter[1] with an alternative syntax for strings. Similar to when you type "Hello World" to a Scheme interpreter and it echoes back the string you typed, BRL reads from the beginning of a file to the first non-escaped [and outputs the resulting string.

In Web Page for Bruce, BRL reads a string starting with the beginning of the file and ending with [. The next thing BRL reads is a compound expression consisting of the three sub-expressions `define`, `myname`, and `"Bruce"`. This doesn't produce any output. Then it reads another define expression. Go back and read that example again, this time looking from an inside-out perspective. Play the role of a BRL interpreter. Read a Scheme expression. Output its value. Repeat. BRL is extremely simple when you know how to look at it.

Advocacy: Other languages can make only very limited use of the text "outside" code delimiters, because it must be converted to print statements. These can be enclosed in loops or other flow-control constructs, but can't be used as function arguments.

[1] The current implementation of BRL is actually a compiler, but this fact is easy to overlook since the compilation step is automatic.

As a result, programmers frequently use quote-and-backslash syntax for strings. Omitted backslashes are a common reason for code not working right the first time. Important spaces are often omitted because it's counterintuitive to put them between a quote mark and the quote-enclosed text, e.g. `" favorite...`. Using BRL speeds development by eliminating these annoying sources of error. With other languages, the best you can hope for is that yet another string-quoting syntax is available and better than either the template syntax or quote-and-backslash.

Other languages are designed by piling feature upon feature. BRL is designed by removing the restrictions on existing features that make additional features appear necessary. There is tremendous power in being able to use BRL's template syntax in any context, not just "top-level" output to the web browser, as subsequent examples will illustrate.

4.1.2 Bruce's Ruse Gets Trickier

Bruce's Web Page is already playing games with visitors. His "favorite number" is either 0 or 1, chosen randomly. Let's take this one step further, adding more sophisticated random content.

```
[(define myname "Bruce")
 (define my "Bruce's")]

<pre>
This is [my] web page.

[(if (zero? (brl-random 2))
     (brl myname] refers to [myname] in 3rd person.[)
     (brl ]Should [myname] run for office?  Yes![))]
</pre>
```

In this example, `(brl-random 2)` returns either 0 or 1. If it returns 0, visitors will see "Bruce refers to Bruce in 3rd person." Otherwise they will see "Should Bruce run for office? Yes!"

Let's use the inside-out perspective to clarify what's happening. We have an `if` expression, which has 3 sub-expressions. The first sub-expression is a predicate, in this case, `(zero? (brl-random 2))`. If the predicate evaluates to true, the second sub-expression is evaluated. If false, the third is evaluated.

The second sub-expression is a `brl` expression. It takes all its sub-expressions and outputs them as part of the web page. Using the inside-out perspective, we see that its sub-expressions are the variable `myname`, the string `" refers to "`, the variable `myname` again, and the string `" in 3rd person."`.

As a performance optimization, `brl` expressions print directly to the web page rather than returning a string to be output. When you need such a string, use `brl-string` instead of `brl`. For example:

```
[(define myface
   (brl-string ]<img src="/img/myface.jpg" alt="[my] face">[)]
```

You can also use the standard Scheme `string-append` in cases like the above, when no non-strings are involved.

Advocacy: HTML coders who know nothing about programming can quickly learn to deal with BRL. Everything outside square brackets is their domain; everything inside is off limits. In the `define myface` example above, width/height attributes could be added. Attributes with quotes could be inserted without backslashes. The HTML coder needn't know five different syntactic conventions for strings, just one. An adventurous HTML coder might even replace `alt="[my] face"` with `alt="[myname]"` and find that it works! BRL truly provides gentle-slope learning for the non-programmer.

Traditional tools require extreme caution by HTML coders working "within" programming code. New tools continue to be written in an attempt to make walls that keep non-programmers away from programming code. However, there will always be a point where programming and non-programming meet. BRL is the best tool for that meeting point.

4.2 Shared variables and common headers/footers

Any definition from "Web Page for Bruce", e.g. `(define myname "Bruce")`, can be cut from that page and put into your '`sitedefs.scm`' file. Then any instance of `[myname]` on *any* page will output `Bruce`. If you don't know where '`sitedefs.scm`' is, put the following code in a BRL page:

```
Look in [brl-scmdir] for sitedefs.scm
```

That page will tell you where to look.

Function definitions go in '`sitedefs.scm`' as well. A convenient way to do BRL development is to create functions in the file you're working on as needed, then later cut/paste useful functions into '`sitedefs.scm`' for use in other pages.

Common headers/footers can be implemented by defining strings in sitedefs.scm, but there is a better way. To output a common header in a BRL page, use the following code:

```
[(paste "/header.brl")]
```

This will insert output from '`header.brl`' (found in the document root, or under *brldir*, depending how BRL is configured at your site). This will *not* introduce variables from '`header.brl`' into your page. Use '`sitedefs.scm`' for that.

In the above example, `"header.brl"` and `"../header.brl"` would work just as they would in an HTML anchor. Use `brl-url-contents` instead of `paste` if you want to paste in output from another server.

4.3 HTML Forms and CGI Environment Variables

```
 1 [
 2  (define-input word)
 3 ]
 4 <html>
 5 <head>
 6 <title>Your word: [word]</title>
 7 </head>
 8
 9 <body>
10 <p>Your word is [word].</p>
11 <form>
```

```
11 Type a word: <input name="word">
13 <input type="Submit">
14 </form>
15 </body>
16 </html>
```

In this example, the `define-input` syntax on line 2 is used to name a variable that will hold the value of an HTML input of the same name. If there had been more than one such input, they would all be included in the same line, e.g. `(define-input word1 word2)`.

If you try out this example, you'll get a form into which you can enter a word. When you submit the form, you'll go to the same page, with the word you typed substituted for `[word]`.

Interestingly, when you first try this example, before you submit any form, you'll see `()` substituted for `[word]`. This is because no form input named `word` was submitted. Form inputs can be submitted with any number of values, e.g. same-name checkboxes. Usually you expect one value, in which case you'll get a string. However, if there are zero, two or more values, you'll get a list.

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. **Validate your inputs** if you plan on doing anything interesting with them.

Below is an example of validating inputs. It's a rather involved example. The source for it is in '`learnbrl.war`'. It will pay to try out this example live as you learn the concepts in it.

```
 1  [
 2   (define (valid w)
 3     (if (string? w)
 4         w
 5         "something"))
 6   (define-input (valid word1) (brl-nonblank? word2) word3)
 7   (define-cgi SERVER_NAME)
 8  ]
 9  <html>
10  <head>
11  <title>Backwards</title>
12  </head>
13
14  <body>
15  <form>
16  Type three words: <input name="word1">
17  <input name="word2"> <input name="word3">
18  <input type="Submit">
19  </form>
20  <p>[word1] spelled backwards is
21     [(list->string (reverse (string->list word1)))]
22  </p>
23  [(brl-when word2 ]
24   <p>Second word: [word2]</p>
25  [)]
```

```
26  <p>Third word: [word3]</p>
27  <p>This message brought to you by [SERVER_NAME] as a public
28  service.</p>
29   [(brl-guess-input brl-context)]
30  </body>
31  </html>
```

In this example, `valid` is a procedure defined in lines 2-5 to validate inputs. For those who skipped the introduction to Scheme, here's what it does: It takes an argument, `w`. If `w` is a string, `w` is returned. If `w` is not a string, return the string `"something"`.

The `define-input` syntax on line 6 passes the `word1` input through the procedure we named `valid` before binding it to the variable `word1`. Later, line 21 can simply assume that `word1` is a string. Note that if you want to use `valid` in other pages, you can cut it from lines 2-5 and paste it into '`sitedefs.scm`'.

Also on line 6, `brl-nonblank?` is used on `word2` to return false (`#f`) if `word2` is an empty list or a blank string. Otherwise it returns `word2`[2] This makes `word2` useful in all kinds of conditionals, including the `(brl-when ...)` code on lines 23-25.

On line 6, we chose to leave `word3` unvalidated, just to show that validating some inputs does not mandate that you validate all. Try changing it to validate with `brl-list` so that it's always a list, even if there's exactly one value. This is useful for same-name checkboxes.

Line 7 illustrates the `define-cgi` syntax. Though we chose not to validate the variable in this example, you can do validation just as with the `define-input` example. Validation is sometimes necessary because of user-supplied data. Even though BRL is not implemented using CGI, it uses this syntax to mimic CGI environment variables

On line 29, `brl-guess-input` returns a list that prints as a valid `define-input` line for all the form inputs received. This is handy if you have a form with lots of inputs. When you first write a BRL page as the action for that form, you don't have to pick out all the input names. Just put `[(brl-guess-input brl-context)]` on the BRL page, submit the form, and use the resulting `define-input` line as your starting point.

Advocacy: Competing systems have two common approaches to handling form inputs. Some, e.g. PHP3, automatically assign variables named according to form inputs. This is makes for convenient coding, but not convenient debugging. There is no way to distinguish a typo from a valid form input name that simply wasn't supplied. In large pages, this can cause difficult-to-track bugs, where an empty string is being used somewhere. Also, having form inputs automatically shadow global variables opens some serious security holes. In PHP4, the variables are no longer assigned automatically. However, you still have the typo issue.

Other systems, e.g. Java servlets, have a special Request object with methods for getting form input values. You assign a variable, e.g. `word=request.getParameter("word");`. This is good because it gives you only one place to make a difficult-to-catch typo. If you later reference `werd` instead of `word`, you'll get an error rather than simply having `werd` used as an empty string.

[2] By convention, Scheme procedures that return only #t or #f have a question mark at the end, and those that return other values do not. To be named consistently, one of `brl-blank?` or `brl-nonblank?` had to break this convention.

BRL takes the optimal middle ground. You have one place where inputs are declared. Typos later will result in errors. However, this one place uses a concise, non-redundant syntax, making it almost as convenient to code as PHP3 minus the security holes, and just as convenient to debug as a servlet.

What's more, the validation syntax encourages developers to write code that's easy to audit for security holes related to user-supplied input.

4.4 Path Segments

Suppose you want, for example, `http://example.com/map/5/6` to represent points on a map with coordinates (5, 6). You'll first need to tell the BRL servlet what file to look at for such URIs. Here's what you'd put in '`sitedefs.scm`', assuming that '`/var/www/html/map.brl`' is the full path to the BRL file:

```
(define (brl-uri-map uri default-file)
  (cond
   ((brl-starts-with? "/map/" uri)
    "/var/www/html/map.brl")
   (else default-file)))
```

You may also need to configure your web server to recognize that such URIs should be passed to the BRL servlet. However, for testing purposes it might be easier to just use `http://example.com/servlet/brlsv/map/5/6` as the URL.

Here is the code you would put in '`map.brl`':

```
[
 (define-path (string->number x) (string->number y))
]
<html> <head><title>Position ([x], [y])</title></head>
<body>
 You are at position ([x], [y]) on the map.
 You may go <a href="[(+ y 1)]">north</a>,
 <a href="[(- y 1)]">south</a>,
 <a href="../[(+ x 1)]/[y]">east</a>,
 <a href="../[(- x 1)]/[y]">west</a>.
</body> </html>
```

On line 2, `define-path` takes the last two path segments of `/map/5/6`, passes them to the `string->number` procedure, and binds the results to variables `x` and `y`. Had we been content to leave them as strings, we could have simply used `(define-path x y)`.

We could also have used `(brl-path-segments brl-context)` to return a list of the three strings, `("map" "5" "6")`. This is more useful than the `define-path` syntax if the number of segments varies.

We can omit line numbers in this example, because only line 2 teaches something new about BRL. The only other thing I'd point out is the HTML title in line 4. It is good to make the title as specific as possible. Most browsers have a menu associated with the Back button to enable users to go back several pages at once. A more specific title makes this menu useful.

Advocacy: Isn't `[x]` a lot more readable than `<%=x%>` would have been in this example? Particularly look at the `<a href="...">` parts.

4.5 Cookies

A web server may pass a name/value pair called a "cookie" back to a web client, with the intent that the client pass that same cookie back to the server from then on. Do not use cookies to keep track of the state of what a particular client is doing. That client might have multiple windows open doing different things, or might have used the back button.

Appropriate uses of cookies would be to remember a user-supplied time zone or postal code, or keeping track of items in a shopping cart. Cookies in BRL are always strings, or the Scheme value `#f` if no cookie by that name is set.

The following example sets a cookie named `c`:

```
1 [(define-cookie c)]
2 <p>The cookie named c has value "[c]".</p>
3 [(set! c (brl-random-typeable 2))]
4 <p>The <a href="">next</a> value will be "[c]".</p>
```

If we were doing anything more interesting with the cookie value than displaying it, we would have used the same validation syntax as with `define-input`.

If the value were `set!` multiple times in that page, only the last value would be sent back to the client as a cookie.

See Section 5.2 [Web Functions], page 35, for more sophisticated ways to get/set cookies.

4.6 Session variables

Sometimes a cookie is almost the right thing, except the value is too large to be a practical cookie value, or the nature of the data makes it hard to validate. In such cases, data can be kept on the server side in what's called a session variable. A cookie is used to associate the client with this data, so all the caveats of the previous section apply. Additionally, session variables are transient, and time out after a certain period, configurable in your servlet engine. Most of the time, information should either be kept client-side or be put in a database on the server side. However, session variables are convenient and sometimes appropriate.

```
 1 [(define-session a b c)]
 2
 3 <p>The session variables are [(list a b c)].</p>
 4
 5 [
 6 (set! a b)
 7 (set! b c)
 8 (set! c (brl-random-typeable 2))
 9 ]
10
11 <p>The session variables <a href="">will be</a> [(list a b c)].</p>
```

As with cookies, these values carry over page to page. If the value is `set!` multiple times, only the last becomes the new session variable value.

Unlike with cookies, session variables are not necessarily strings. They can be numbers, lists, vectors, serializable Java objects, etc. There is no validation syntax in `define-session`. It is not user-supplied data.

With the syntax `(define-session (ses) a b c)` we could have bound the variable `ses` to the session object, allowing more sophisticated operations, such as retrieving session variables whose names conflict with input names or other local variables. See Section 5.2 [Web Functions], page 35.

The most common misuse of session variables is in multi-step forms. To accumulate values from previous forms as you go along, use hidden form inputs instead. For example, `[(brl-html-hidden a b c)]` would expand to the following html:

```
<input type="hidden" name="a" value="a's value">
<input type="hidden" name="b" value="b's value">
<input type="hidden" name="c" value="c's value">
```

4.7 Sending e-mail

The definition below goes in '`sitedefs.scm`' to set things up to send mail via an SMTP server. If you are using BRL Cabaret or some other site that has already been set up, this definition is already there for you. You do not have to include it in an individual page; it is a site-wide definition (ergo, '`sitedefs.scm`').

```
(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.

Obviously, you can define several such procedures with any names you like, but always include a default procedure named mail to ease portability of BRL code from site to site.

The following page shows how to use the mail procedure. It would be the ACTION page for an HTML form:

```
 1  <html>
 2  <head><title>Mail sent</title></head>
 3
 4  <body>
 5  [(define-input email yourname quest colour)
 6
 7   (mail (list "strangeman@example.com")
 8  ]From: [email]
 9  To: strangeman@example.com
10  Subject: questions three
11
12  What is your name?              [yourname]
13  What is your quest?             [quest]
14  What is your favourite colour?  [colour]
15  [)]
```

```
16
17  <p>Your e-mail has been sent.  Thank you.</p>
18
19  </body>
20  </html>
```

Line 5 defines the inputs that come from the HTML form. Line 7 invokes the mail procedure with a list of one recipient. Lines 8-14 define the body of the e-mail message. It is important that there be no spaces between] and the first header (line 8), and that there is a blank line after the headers (line 11). Line 15 has the close paren for the mail expression.

Let's look at this example from the perspective discussed earlier (see Section 4.1.1 [Understanding BRL Inside-Out], page 14). Don't look at square brackets as enclosing `[code]`. Look at them in the reverse sense, enclosing `]strings[` to see what's happening. The first argument to `mail` on line 7 is a list. The second argument, on line 8, is the string `"From: "`, followed by the variable `email` as the third argument. The fourth argument is a string that starts on line 8 with a newline character, spans several lines, and ends just before the variable `yourname` (the fifth argument, on line 12).

The `mail` procedure, as explained earlier, concatenates its second and subsequent arguments to create an Internet message, which it then sends via SMTP. Taking the inside-out perspective, the closing paren on line 15 makes perfect sense. The square bracket before it ends a string that is part of the e-mail message, since it's an argument to `mail`. The square bracket after the close paren begins a string that will be sent back to the browser because it's at the top level, i.e. not being used as an argument to anything.

Advocacy: In 1994, cgiemail was developed at the Massachusetts Institute of Technology to allow non-programmers to easily specify what e-mail messages sent from their HTML forms should look like using a simple template syntax. Lines 8-14 look like that syntax. A non-programmer could easily customize the example above. BRL is almost as simple as cgiemail, but much more powerful.

Only BRL uses the same friendly template syntax for constructing e-mail messages as is used for constructing web pages.

4.8 SQL in BRL

4.8.1 One-Time Preparation for All Pages

If you are using BRL Cabaret, the table and JDBC driver are already set up. You may skip ahead to Section 4.8.2 [Queries in an Individual Page], page 23.

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 PostgreSQL driver:

```
(sql-driver "org.postgresql.Driver")
(define (db1 brl-context)
  (brl-sql-connection brl-context
                      "jdbc:postgresql://localhost/db-here"
                      "user-here"
                      "password-here"))

(define ss brl-mysql-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-mysql-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 or PostgreSQL (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`'

Advocacy: Simple examples for competing systems generally put database info (driver, server, user, etc.) directly in the page. Only with BRL is it natural and easy to centralize, while still maintaining the flexibility of having many different DBs to connect to.

Unclosed database objects (connections, statements, etc) are a major headache for server-side web programming. Sometimes this happens because of an uncaught error. Sometimes a programmer simply forgets to close the object when done. BRL eliminates the problem in both cases.

4.8.2 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[3] that can be used repeatedly for SQL queries on that connection.

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.

4.8.3 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>
```

[3] It is currently implemented as JDBC's `java.sql.Statement` object.

```
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   (define-input 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 4.8.1 [One-Time Preparation for All Pages], page 22.

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.

4.8.4 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       (brl-when (group-ending? color)
17                 #\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` within 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.

4.8.5 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.

4.8.6 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[4] 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  (define-input color name order1 order2)
5  (define where-clause
6    (string-append "where 1=1"
7      (if (brl-nonblank? name)
```

[4] more precisely, activating the link

```
 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 (list "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 (list "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
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.

4.9 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.

4.9.1 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.

4.9.2 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.

4.9.3 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"`.

4.10 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 equivalents or approximations.

`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-all-blank?`
: Takes any number of arguments and returns true only if all are blank.

`brl-any-blank?`
: Takes any number of arguments and returns true if any are blank.

`brl-all-nonblank?`
: Takes any number of arguments and returns true only if none are blank.

`brl-any-nonblank?`
: Takes any number of arguments and returns true if any are not blank.

`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-content-type!`
: takes a BRL context argument and a string representing a MIME type. Sets the Content-Type header to be delivered to the browser.

`brl-http-redirect!`
: takes a BRL context argument and a string containing an absolute URL. The browser is redirected to that URL.

`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-http-header!`
: takes a BRL context argument, an HTTP header name and a value. The HTTP header of that name is set to that value.

`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.

`brl-session`
: Takes a BRL context argument and returns a procedure of one or two arguments. The returned procedure, when passed one argument (a symbol), will return the value associated with that symbol for this HTTP session, or #f if said symbol is unbound. When passed two arguments, a symbol and any object, it will associate the object with the symbol for later retrieval within the same session.

`brl-session-id`
: When passed a procedure returned by `brl-session`, `brl-session-id` returns a unique string identifying the session.

`brl-session-new?`
: When passed a procedure returned by `brl-session`, `brl-session-new?` returns true if this session is being created with the current request, i.e. no valid session cookie was sent from the client, so the server will assign one.

`brl-session-invalidate!`
: When passed a procedure returned by `brl-session`, `brl-session-invalidate!` will keep any state associated with this session from being retrieved for subsequent HTTP requests. This procedure should be used to log a user out of a session-based application.

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 Network Functions

`brl-tcp-socket`
: takes a string representing a hostname and an integer representing a port. Returns a connection for use with `brl-tcp-in` and `brl-tcp-out`.

`brl-tcp-in`
: takes a tcp connection as returned by `brl-tcp-socket` and returns a input port to receive data from the remote host.

`brl-tcp-out`
: takes a tcp connection as returned by `brl-tcp-socket` and returns a output port to send data to the remote host.

`brl-smtp-date`
: takes a date argument and returns a string representing that date, formatted for use with SMTP. The same format is useful with HTTP.

`brl-smtp-sender`
: takes three strings as arguments. The first is a mail host. The second is the hostname the local host will use to identify itself to the mail host. The third is an "envelope sender" address. A procedure is returned that takes two or more arguments. The first argument is a list of recipients. Subsequent arguments are made into a string forming the headers, a blank line, and the body of an e-mail message.

5.5 Miscellaneous Functions

`brl`
: This syntax causes its arguments to be evaluated left-to-right. The resulting values (if applicable) are displayed.

`brl-when`
: This syntax is convenient shorthand for `(if arg1 (brl arg2 ...) "")`.

`brl-unless`
: This syntax is convenient shorthand for `(if arg1 "" (brl arg2 ...))`.

`silent`
: This syntax is like Scheme's `begin`, but produces no output when used in a BRL page.

`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.

6 Regular Expressions

Starting with release 2.1.22, BRL includes Dorai Sitaram's '`pregexp.scm`' package, which provides support for Perl 5.x-style regular expressions. This chapter is taken from the documentation in that package.

6.1 Regexp Introduction

A *regexp* is a string that describes a pattern. A regexp matcher tries to *match* this pattern against (a portion of) another string, which we will call the *text string*. The text string is treated as raw text and not as a pattern.

Most of the characters in a regexp pattern are meant to match occurrences of themselves in the text string. Thus, the pattern `"abc"` matches a string that contains the characters `a`, `b`, `c` in succession.

In the regexp pattern, some characters act as *metacharacters*, and some character sequences act as *metasequences*. That is, they specify something other than their literal selves. For example, in the pattern `"a.c"`, the characters `a` and `c` do stand for themselves but the *metacharacter* `.` can match *any* character (other than newline). Therefore, the pattern `"a.c"` matches an `a`, followed by *any* character, followed by a `c`.

If we needed to match the character `.` itself, we *escape* it, ie, precede it with a backslash (`\`). The character sequence `\.` is thus a *metasequence*, since it doesn't match itself but rather just `.`. So, to match `a` followed by a literal `.` followed by `c`, we use the regexp pattern `"a\\.c"`.[1] Another example of a metasequence is `\t`, which is a readable way to represent the tab character.

We will call the string representation of a regexp the *U-regexp*, where *U* can be taken to mean *Unix-style* or *universal*, because this notation for regexps is universally familiar. Our implementation uses an intermediate tree-like representation called the *S-regexp*, where *S* can stand for *Scheme*, *symbolic*, or *s-expression*. S-regexps are more verbose and less readable than U-regexps, but they are much easier for Scheme's recursive procedures to navigate.

6.2 Regexp procedures provided

'`pregexp.scm`' provides the user with the procedures `pregexp`, `pregexp-match-positions`, `pregexp-match`, `pregexp-replace`, and `pregexp-replace*`. All the identifiers introduced by '`pregexp.scm`' have the prefix `pregexp`, so they are unlikely to clash with other names in Scheme, including those of any natively provided regexp operators.

[1] The double backslash is an artifact of Scheme strings, not the regexp pattern itself. When we want a literal backslash inside a Scheme string, we must escape it so that it shows up in the string at all. Scheme strings use backslash as the escape character, so we end up with two backslashes — one Scheme-string backslash to escape the regexp backslash, which then escapes the dot. Another character that would need escaping inside a Scheme string is `"`.

6.2.1 pregexp

The procedure `pregexp` takes a U-regexp, which is a string, and returns an S-regexp, which is a tree.

```
(pregexp "c.r")
=> (:sub (:or (:seq #\c :any #\r)))
```

There is rarely any need to look at the S-regexps returned by `pregexp`.

6.2.2 pregexp-match-positions

The procedure `pregexp-match-positions` takes a regexp pattern and a text string, and returns a *match* if the pattern *matches* the text string. The pattern may be either a U- or an S-regexp. (`pregexp-match-positions` will internally compile a U-regexp to an S-regexp before proceeding with the matching. If you find yourself calling `pregexp-match-positions` repeatedly with the same U-regexp, it may be advisable to explicitly convert the latter into an S-regexp once beforehand, using `pregexp`, to save needless recompilation.)

`pregexp-match-positions` returns `#f` if the pattern did not match the string; and a list of *index pairs* if it did match. Eg,

```
(pregexp-match-positions "brain" "bird")
=> #f

(pregexp-match-positions "needle" "hay needle stack")
=> ((4 . 10))
```

In the second example, the integers 4 and 10 identify the substring that was matched. 1 is the starting (inclusive) index and 2 the ending (exclusive) index of the matching substring.

```
(substring "hay needle stack" 4 10)
=> "needle"
```

Here, `pregexp-match-positions`'s return list contains only one index pair, and that pair represents the entire substring matched by the regexp. When we discuss *subpatterns* later, we will see how a single match operation can yield a list of *submatches*.

`pregexp-match-positions` takes optional third and fourth arguments that specify the indices of the text string within which the matching should take place.

```
(pregexp-match-positions "needle"
  "his hay needle stack -- my hay needle stack -- her hay needle stack"
  24 43)
=> ((31 . 37))
```

Note that the returned indices are still reckoned relative to the full text string.

6.2.3 pregexp-match

The procedure `pregexp-match` is called like `pregexp-match-positions` but instead of returning index pairs it returns the matching substrings:

```
(pregexp-match "brain" "bird")
=> #f

(pregexp-match "needle" "hay needle stack")
```

```
=> ("needle")
```

`pregexp-match` also takes optional third and fourth arguments, with the same meaning as does `pregexp-match-positions`.

6.2.4 pregexp-replace

The procedure `pregexp-replace` replaces the matched portion of the text string by another string. The first argument is the regexp, the second the text string, and the third is the *insert string* (string to be inserted).

```
(pregexp-replace "te" "liberte" "ty")
=> "liberty"
```

6.2.5 pregexp-replace*

The procedure `pregexp-replace*` replaces *all* matches in the text string by the insert string:

```
(pregexp-replace* "te" "liberte egalite fraternite" "ty")
=> "liberty egality fratyrnity"
```

6.3 The regexp pattern language

Here is a complete description of the regexp pattern language recognized by the `pregexp` procedures.

6.3.1 Basic assertions

The *assertions* `^` and `$` identify the beginning and the end of the text string respectively. They ensure that their adjoining regexps match at one or other end of the text string. Examples:

```
(pregexp-match-positions "^contact" "first contact")
=> #f
```

The regexp fails to match because `contact` does not occur at the beginning of the text string.

```
(pregexp-match-positions "laugh$" "laugh laugh laugh laugh")
=> ((18 . 23))
```

The regexp matches the *last* `laugh`.

The metasequence `\b` asserts that a *word boundary* exists.

```
(pregexp-match-positions "yack\\b" "yackety yack")
=> ((8 . 12))
```

The `yack` in `yackety` doesn't end at a word boundary so it isn't matched. The second `yack` does and is.

The metasequence `\B` has the opposite effect to `\b`. It asserts that a word boundary does not exist.

```
(pregexp-match-positions "an\\B" "an analysis")
=> ((3 . 5))
```

The `an` that doesn't end in a word boundary is matched.

6.3.2 Characters and character classes

Typically a character in the regexp matches the same character in the text string. Sometimes it is necessary or convenient to use a regexp metasequence to refer to a single character. Thus, metasequences `\n`, `\r`, `\t`, and `\.` match the newline, return, tab and period characters respectively.

The *metacharacter* period (`.`) matches *any* character other than newline.

```
(pregexp-match "p.t" "pet")
=> ("pet")
```

It also matches `pat`, `pit`, `pot`, `put`, and `p8t` but not `peat` or `pffft`.

A *character class* `[...]` matches any one character from the set enclosed within the brackets. Thus `"p[aeiou]t"` matches `pat`, `pet`, `pit`, `pot`, `put` and nothing else.

Inside the brackets, a hyphen (`-`) between two characters specifies the ascii range between the characters. Eg, `"ta[b-dgn-p]"` matches `tab`, `tac`, `tad`, *and* `tag`, *and* `tan`, `tao`, `tap`.

An initial caret (`^`) after the left bracket inverts the set specified by the rest of the contents, ie, it specifies the set of characters *other than* those identified in the brackets. Eg, `"do[^g]"` matches all three-character sequences starting with `do` except `dog`.

Note that the metacharacter `^` inside brackets means something quite different from what it means outside. Most other metacharacters (`.`, `*`, `+`, `?`, etc) cease to be metacharacters when inside brackets, although you may still escape them for peace of mind. `-` is a metacharacter only when it's inside brackets.

6.3.2.1 Some frequently used character classes

Some standard character classes can be conveniently represented as metasequences instead of as explicit bracketed expressions. `\d` matches a digit (`[0-9]`); `\s` matches a whitespace character; and `\w` matches a character that could be part of a "word".[2]

The upper-case versions of these metasequences stand for the inversions of the corresponding character classes. Thus `\D` matches a non-digit, `\S` a non-whitespace character, and `\W` a non-"word" character.

Remember to include a double backslash when putting these metasequences in a Scheme string:

```
(pregexp-match "\\d\\d"
  "0 dear, 1 have 2 read catch 22 before 9")
=> ("22")
```

These character classes can be used inside a bracketed expression. Eg, `"[a-z\\d]"` matches a lower-case letter or a digit.

6.3.2.2 POSIX character classes

A *POSIX character class* is a special metasequence of the form `[:...:]` that can be used only inside a bracketed expression. The POSIX classes supported are

[2] Following regexp custom, we identify "word" characters as `[A-Za-z0-9_]`, although these are too restrictive for what a Schemer might consider a "word".

`[:alnum:]`
: letters and digits

`[:alpha:]`
: letters

`[:algor:]`
: the letters `c`, `h`, `a` and `d`

`[:ascii:]`
: 7-bit ascii characters

`[:blank:]`
: widthful whitespace, ie, space and tab

`[:cntrl:]`
: "control" characters, viz, those with code < 32

`[:digit:]`
: digits, same as `\d`

`[:graph:]`
: characters that use ink

`[:lower:]`
: lower-case letters

`[:print:]`
: ink-users plus widthful whitespace

`[:space:]`
: whitespace, same as `\s`

`[:upper:]`
: upper-case letters

`[:word:]` letters, digits, and underscore, same as `\w`

`[:xdigit:]`
: ex digits

For example, the regexp `"[[:alpha:]_]"` matches a letter or underscore.

```
(pregexp-match "[[:alpha:]_]" "--x--")
=> ("x")

(pregexp-match "[[:alpha:]_]" "--_--")
=> ("_")

(pregexp-match "[[:alpha:]_]" "--:--")
=> #f
```

The POSIX class notation is valid *only* inside a bracketed expression. For instance, `[:alpha:]`, when not inside a bracketed expression, will *not* be read as the letter class. Rather it is (from previous principles) the character class containing the characters `:`, `a`, `l`, `p`, `h`.

```
(pregexp-match "[:alpha:]" "--a--")
=> ("a")

(pregexp-match "[:alpha:]" "--_--")
=> #f
```

By placing a caret (^) immediately after [:, you get the inversion of that POSIX character class. Thus, [:^alpha] is the class containing all characters except the letters.

6.3.3 Quantifiers

The *quantifiers* *, +, and ? match respectively: zero or more, one or more, and zero or one instances of the preceding subpattern.

```
(pregexp-match-positions "c[ad]*r" "cadaddadddr")
=> ((0 . 11))
(pregexp-match-positions "c[ad]*r" "cr")
=> ((0 . 2))

(pregexp-match-positions "c[ad]+r" "cadaddadddr")
=> ((0 . 11))
(pregexp-match-positions "c[ad]+r" "cr")
=> #f

(pregexp-match-positions "c[ad]?r" "cadaddadddr")
=> #f
(pregexp-match-positions "c[ad]?r" "cr")
=> ((0 . 2))
(pregexp-match-positions "c[ad]?r" "car")
=> ((0 . 3))
```

6.3.3.1 Numeric quantifiers

You can use braces to specify much finer-tuned quantification than is possible with *, +, ?.

The quantifier {m} matches *exactly* m instances of the preceding *subpattern*. m must be a nonnegative integer.

The quantifier {m,n} matches at least m and at most n instances. m and n are nonnegative integers with m <= n. You may omit either or both numbers, in which case m defaults to 0 and n to infinity.

It is evident that + and ? are abbreviations for {1,} and {0,1} respectively. * abbreviates {,}, which is the same as {0,}.

```
(pregexp-match "[aeiou]{3}" "vacuous")
=> ("uou")

(pregexp-match "[aeiou]{3}" "evolve")
=> #f

(pregexp-match "[aeiou]{2,3}" "evolve")
```

```
=> #f

(pregexp-match "[aeiou]{2,3}" "zeugma")
=> ("eu")
```

6.3.3.2 Non-greedy quantifiers

The quantifiers described above are *greedy*, ie, they match the maximal number of instances that would still lead to an overall match for the full pattern.

```
(pregexp-match "<.*>" "<tag1> <tag2> <tag3>")
=> ("<tag1> <tag2> <tag3>")
```

To make these quantifiers *non-greedy*, append a `?` to them. Non-greedy quantifiers match the minimal number of instances needed to ensure an overall match.

```
(pregexp-match "<.*?>" "<tag1> <tag2> <tag3>")
=> ("<tag1>")
```

The non-greedy quantifiers are respectively: `*?`, `+?`, `??`, `{m}?`, `{m,n}?`. Note the two uses of the metacharacter `?`.

6.3.4 Clusters

Clustering, ie, enclosure within parens `(...)`, identifies the enclosed *subpattern* as a single entity. It causes the matcher to *capture* the *submatch*, or the portion of the string matching the subpattern, in addition to the overall match.

```
(pregexp-match "([a-z]+) ([0-9]+), ([0-9]+)" "jan 1, 1970")
=> ("jan 1, 1970" "jan" "1" "1970")
```

Clustering also causes a following quantifier to treat the entire enclosed subpattern as an entity.

```
(pregexp-match "(poo )*" "poo poo platter")
=> ("poo poo " "poo ")
```

The number of submatches returned is always equal to the number of subpatterns specified in the regexp, even if a particular subpattern happens to match more than one substring or no substring at all.

```
(pregexp-match "([a-z ]+;)*" "lather; rinse; repeat;")
=> ("lather; rinse; repeat;" " repeat;")
```

Here the `*`-quantified subpattern matches three times, but it is the last submatch that is returned.

It is also possible for a quantified subpattern to fail to match, even if the overall pattern matches. In such cases, the failing submatch is represented by `#f`.

```
(define date-re
  ;match ‘month year’ or ‘month day, year’.
  ;subpattern matches day, if present
  (pregexp "([a-z]+) +([0-9]+,)? *([0-9]+)"))

(pregexp-match date-re "jan 1, 1970")
=> ("jan 1, 1970" "jan" "1," "1970")
```

```
(pregexp-match date-re "jan 1970")
=> ("jan 1970" "jan" #f "1970")
```

6.3.4.1 Backreferences

Submatches can be used in the insert string argument of the procedures `pregexp-replace` and `pregexp-replace*`. The insert string can use `\n` as a *backreference* to refer back to the *n*th submatch, ie, the substring that matched the *n*th subpattern. `\0` refers to the entire match, and it can also be specified as `\&`.

```
(pregexp-replace "_(.+?)_"
  "the _nina_, the _pinta_, and the _santa maria_"
  "*\\1*")
=> "the *nina*, the _pinta_, and the _santa maria_"

(pregexp-replace* "_(.+?)_"
  "the _nina_, the _pinta_, and the _santa maria_"
  "*\\1*")
=> "the *nina*, the *pinta*, and the *santa maria*"

;recall: \S stands for non-whitespace character

(pregexp-replace "(\\S+) (\\S+) (\\S+)"
  "eat to live"
  "\\3 \\2 \\1")
=> "live to eat"
```

Use `\\` in the insert string to specify a literal backslash. Also, `\$` stands for an empty string, and is useful for separating a backreference `\n` from an immediately following number.

Backreferences can also be used within the regexp pattern to refer back to an already matched subpattern in the pattern. `\n` stands for an exact repeat of the *n*th submatch.[3]

```
(pregexp-match "([a-z]+) and \\1"
  "billions and billions")
=> ("billions and billions" "billions")
```

Note that the backreference is not simply a repeat of the previous subpattern. Rather it is a repeat of *the particular substring already matched by the subpattern.*

In the above example, the backreference can only match `billions`. It will not match `millions`, even though the subpattern it harks back to — `([a-z]+)` — would have had no problem doing so:

```
(pregexp-match "([a-z]+) and \\1"
  "billions and millions")
=> #f
```

The following corrects doubled words:

```
(pregexp-replace* "(\\S+) \\1"
  "now is the the time for all good men to to come to the aid of of the party"
  "\\1")
```

[3] `\0`, which is useful in an insert string, makes no sense within the regexp pattern, because the entire regexp has not matched yet that you could refer back to it.

```
=> "now is the time for all good men to come to the aid of the party"
```

The following marks all immediately repeating patterns in a number string:

```
(pregexp-replace* "(\\d+)\\1"
  "123340983242432420980980234"
  "{\\1,\\1}")
=> "12{3,3}40983{24,24}3242{098,098}0234"
```

6.3.4.2 Non-capturing clusters

It is often required to specify a cluster (typically for quantification) but without triggering the capture of submatch information. Such clusters are called *non-capturing*. In such cases, use `(?:` instead of `(` as the cluster opener. In the following example, the non-capturing cluster eliminates the "directory" portion of a given pathname, and the capturing cluster identifies the basename.

```
(pregexp-match "^(?:[a-z]*/)*([a-z]+)$"
  "/usr/local/bin/mzscheme")
=> ("/usr/local/bin/mzscheme" "mzscheme")
```

6.3.4.3 Cloisters

The location between the `?` and the `:` of a non-capturing cluster is called a *cloister*.[4] You can put *modifiers* there that will cause the enclustered subpattern to be treated specially. The modifier `i` causes the subpattern to match *case-insensitively*:

```
(pregexp-match "(?i:hearth)" "HeartH")
=> ("HeartH")
```

The modifier `x` causes the subpattern to match *space-insensitively*, ie, spaces and comments within the subpattern are ignored. Comments are introduced as usual with a semicolon (`;`) and extend till the end of the line. If you need to include a literal space or semicolon in a space-insensitized subpattern, escape it with a backslash.

```
(pregexp-match "(?x: a   lot)" "alot")
=> ("alot")

(pregexp-match "(?x: a  \\  lot)" "a lot")
=> ("a lot")

(pregexp-match "(?x:
   a \\ man  \\; \\   ; ignore
   a \\ plan \\; \\   ; me
   a \\ canal         ; completely
   )"
 "a man; a plan; a canal")
=> ("a man; a plan; a canal")
```

The global variable `*pregexp-comment-char*` contains the comment character (`#\;`). For Perl-like comments,

[4] A useful, if terminally cute, coinage from the abbots of Perl [Programming Perl].

```
(set! *pregexp-comment-char* #\#)
```

You can put more than one modifier in the cloister.

```
(pregexp-match "(?ix:
   a \\ man  \\; \\   ; ignore
   a \\ plan \\; \\   ; me
   a \\ canal         ; completely
   )"
 "A Man; a Plan; a Canal")
=> ("A Man; a Plan; a Canal")
```

A minus sign before a modifier inverts its meaning. Thus, you can use `-i` and `-x` in a *subcluster* to overturn the insensitivities caused by an enclosing cluster.

```
(pregexp-match "(?i:the (?-i:TeX)book)"
  "The TeXbook")
=> ("The TeXbook")
```

This regexp will allow any casing for `the` and `book` but insists that `TeX` not be differently cased.

6.3.5 Alternation

You can specify a list of *alternate* subpatterns by separating them by '`|`'. The '`|`' separates subpatterns in the nearest enclosing cluster (or in the entire pattern string if there are no enclosing parens).

```
(pregexp-match "f(ee|i|o|um)" "a small, final fee")
=> ("fi" "i")

(pregexp-replace* "([yi])s(e[sdr]?|ing|ation)"
   "it is energising to analyse an organisation
   pulsing with noisy organisms"
   "\\1z\\2")
=> "it is energizing to analyze an organization
   pulsing with noisy organisms"
```

Note again that if you wish to use clustering merely to specify a list of alternate subpatterns but do not want the submatch, use `(?:` instead of `(`.

```
(pregexp-match "f(?:ee|i|o|um)" "fun for all")
=> ("fo")
```

An important thing to note about alternation is that the leftmost matching alternate is picked regardless of its length. Thus, if one of the alternates is a prefix of a later alternate, the latter may not have a chance to match.

```
(pregexp-match "call|call-with-current-continuation"
  "call-with-current-continuation")
=> ("call")
```

To allow the longer alternate to have a shot at matching, place it before the shorter one:

```
(pregexp-match "call-with-current-continuation|call"
  "call-with-current-continuation")
=> ("call-with-current-continuation")
```

In any case, an overall match for the entire regexp is always preferred to an overall nonmatch. In the following, the longer alternate still wins, because its preferred shorter prefix fails to yield an overall match.

```
(pregexp-match "(?:call|call-with-current-continuation) constrained"
  "call-with-current-continuation constrained")
=> ("call-with-current-continuation constrained")
```

6.3.6 Backtracking

We've already seen that greedy quantifiers match the maximal number of times, but the overriding priority is that the overall match succeed. Consider

```
(pregexp-match "a*a" "aaaa")
```

The regexp consists of two subregexps, `a*` followed by `a`. The subregexp `a*` cannot be allowed to match all four `a`'s in the text string `"aaaa"`, even though `*` is a greedy quantifier. It may match only the first three, leaving the last one for the second subregexp. This ensures that the full regexp matches successfully.

The regexp matcher accomplishes this via a process called *backtracking*. The matcher tentatively allows the greedy quantifier to match all four `a`'s, but then when it becomes clear that the overall match is in jeopardy, it *backtracks* to a less greedy match of *three* `a`'s. If even this fails, as in the call

```
(pregexp-match "a*aa" "aaaa")
```

the matcher backtracks even further. Overall failure is conceded only when all possible backtracking has been tried with no success.

Backtracking is not restricted to greedy quantifiers. Nongreedy quantifiers match as few instances as possible, and progressively backtrack to more and more instances in order to attain an overall match. There is backtracking in alternation too, as the more rightward alternates are tried when locally successful leftward ones fail to yield an overall match.

6.3.6.1 Disabling backtracking

Sometimes it is efficient to disable backtracking. For example, we may wish to *commit* to a choice, or we know that trying alternatives is fruitless. A nonbacktracking regexp is enclosed in `(?>...)`.

```
(pregexp-match "(?>a+)." "aaaa")
=> #f
```

In this call, the subregexp `?>a*` greedily matches all four `a`'s, and is denied the opportunity to backpedal. So the overall match is denied. The effect of the regexp is therefore to match one or more `a`'s followed by something that is definitely non-`a`.

6.3.7 Looking ahead and behind

You can have assertions in your pattern that look *ahead* or *behind* to ensure that a subpattern does or does not occur. These "look around" assertions are specified by putting the subpattern checked for in a cluster whose leading characters are: `?=` (for positive lookahead), `?!` (negative lookahead), `?<=` (positive lookbehind), `?<!` (negative lookbehind). Note that the subpattern in the assertion does not generate a match in the final result. It merely allows or disallows the rest of the match.

6.3.7.1 Lookahead

Positive lookahead (`?=`) peeks ahead to ensure that its subpattern *could* match.

```
(pregexp-match-positions "grey(?=hound)"
  "i left my grey socks at the greyhound")
=> ((28 . 32))
```

The regexp `"grey(?=hound)"` matches `grey`, but *only* if it is followed by `hound`. Thus, the first `grey` in the text string is not matched.

Negative lookahead (`?!`) peeks ahead to ensure that its subpattern could not possibly match.

```
(pregexp-match-positions "grey(?!hound)"
  "the gray greyhound ate the grey socks")
=> ((27 . 31))
```

The regexp `"grey(?!hound)"` matches `grey`, but only if it is *not* followed by `hound`. Thus the `grey` just before `socks` is matched.

6.3.7.2 Lookbehind

Positive lookbehind (`?<=`) checks that its subpattern *could* match immediately to the left of the current position in the text string.

```
(pregexp-match-positions "(?<=grey)hound"
  "the hound in the picture is not a greyhound")
=> ((38 . 43))
```

The regexp `(?<=grey)hound` matches `hound`, but only if it is preceded by `grey`.

Negative lookbehind (`?<!`) checks that its subpattern could not possibly match immediately to the left.

```
(pregexp-match-positions "(?<!grey)hound"
  "the greyhound in the picture is not a hound")
=> ((38 . 43))
```

The regexp `(?<!grey)hound` matches `hound`, but only if it is *not* preceded by `grey`.

Lookaheads and lookbehinds can be convenient when they are not confusing.

6.4 An extended example

Here's an extended example from Friedl [Mastering Regular Expressions, p123] that covers many of the features described above. The problem is to fashion a regexp that will match any and only IP addresses or *dotted quads*, ie, four numbers separated by three dots, with each number between 0 and 255. We will use the commenting mechanism to build the final regexp with clarity. First, a subregexp `n0-255` that matches 0 through 255.

```
(define n0-255
  "(?x:
  \\d           ;  0 through   9
  | \\d\\d      ; 00 through  99
  | [01]\\d\\d ;000 through 199
  | 2[0-4]\\d  ;200 through 249
  | 25[0-5]    ;250 through 255
```

```
  )")
```

The first two alternates simply get all single- and double-digit numbers. Since 0-padding is allowed, we need to match both 1 and 01. We need to be careful when getting 3-digit numbers, since numbers above 255 must be excluded. So we fashion alternates to get 000 through 199, then 200 through 249, and finally 250 through 255.[5]

An IP-address is a string that consists of four `n0-255`s with three dots separating them.

```
(define ip-re1
  (string-append
    "^"         ;nothing before
    n0-255      ;the first n0-255,
    "(?x:"      ;then the subpattern of
    "\\."       ;a dot followed by
    n0-255      ;an n0-255,
    ")"         ;which is
    "{3}"       ;repeated exactly 3 times
    "$"         ;with nothing following
    ))
```

Let's try it out.

```
(pregexp-match ip-re1
  "1.2.3.4")
=> ("1.2.3.4")

(pregexp-match ip-re1
  "55.155.255.265")
=> #f
```

which is fine, except that we also have

```
(pregexp-match ip-re1
  "0.00.000.00")
=> ("0.00.000.00")
```

All-zero sequences are not valid IP addresses! Lookahead to the rescue. Before starting to match `ip-re1`, we look ahead to ensure we don't have all zeros. We could use positive lookahead to ensure there *is* a digit other than zero.

```
(define ip-re
  (string-append
    "(?=.*[1-9])" ;ensure there's a non-0 digit
    ip-re1))
```

Or we could use negative lookahead to ensure that what's ahead isn't composed of *only* zeros and dots.

```
(define ip-re
  (string-append
    "(?![0.]*$)" ;not just zeros and dots
                 ;(note: dot is not metachar inside [])
```

[5] Note that `n0-255` lists prefixes as preferred alternates, something we cautioned against (see Section 6.3.5 [Alternation], page 52). However, since we intend to anchor this subregexp explicitly to force an overall match, the order of the alternates does not matter.

```
        ip-re1))
```

The regexp `ip-re` will match all and only valid IP addresses.

```
(pregexp-match ip-re
  "1.2.3.4")
=> ("1.2.3.4")

(pregexp-match ip-re
  "0.0.0.0")
=> #f
```

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 4.7 [Sending e-mail], page 21). A relatively small investment of time acclimating to BRL syntax will repay handsomely in a powerfully expressive tool for writing dynamic web pages.

Index

A

apply 10

B

begin 26
brl 15, 24, 40
brl-all-blank? 34
brl-all-nonblank? 34
brl-any-blank? 34
brl-any-nonblank? 34
brl-blank? 34
brl-content-type! 35
brl-context 23
brl-context-cookies 35
brl-cookie-set! 35
brl-cookie-value 35
brl-decimal-formatter 31, 33
brl-ends-with? 34
brl-format 30, 31, 33
brl-get-update 36
brl-guess-input 18
brl-hash 40
brl-hash-contains-key? 41
brl-hash-get 41
brl-hash-keys 41
brl-hash-put! 40
brl-hash-remove! 41
brl-hash-size 40
brl-hash? 40
brl-html-escape 27, 33
brl-html-hidden 21
brl-html-options 36
brl-http-header! 35
brl-http-redirect! 35
brl-http-status! 35
brl-implementation-version 40
brl-latex-escape 34
brl-list 18
brl-msft-escape 33
brl-nonblank? 18, 34
brl-nonblanks 34
brl-now 31
brl-path-segments 19
brl-prepend-endproc! 32
brl-random 41
brl-random-typeable 41
brl-readall 40
brl-referer-check 25, 35
brl-referer-ok? 35
brl-scheme-escape 33
brl-session 36
brl-session-id 36
brl-session-invalidate! 37
brl-session-new? 36
brl-simple-date-formatter 31, 33
brl-smtp-date 40
brl-smtp-sender 40
brl-split 31, 34
brl-sql-connection 23, 37
brl-sql-escape 33
brl-sql-number 34
brl-sql-statement 24, 37
brl-sql-string 26, 34
brl-starts-with? 34
brl-string 15, 30, 33
brl-string-escaper 33
brl-string-join 31, 35
brl-tcp-in 39
brl-tcp-out 40
brl-tcp-socket 39
brl-trim 31, 34
brl-unless 40
brl-url-arg 36
brl-url-arg-seq 30
brl-url-args 28, 36
brl-url-contents 36
brl-url-escape 28
brl-when 40
brldir 4

C

car 9
cdr 9
CLASSPATH 4
cons 9

D

define 7
define-cgi 18
define-cookie 20
define-input 17
define-path 19
define-session 20

G

group-beginning? 27
group-ending? 27

I

if 9
invoke 32

M

make 32
map 10

N

number->string 30

P

pregexp 44
pregexp-match 44
pregexp-match-positions 44
pregexp-replace 45
pregexp-replace* 45

S

scmdir 4
scmuri 4
silent 40
sitedefs.scm 16
sql-connection 37
sql-connection-close 37
sql-connection? 37
sql-driver 22, 37
sql-execute 38
sql-execute-query 38
sql-execute-update 25, 38
sql-order-desc? 39
sql-order-eqv-di? 39
sql-order-member 39
sql-order-prepend 30, 39
sql-order-reverse 39
sql-partial-update 39
sql-repeat 24, 39
sql-repeat-rsmd 38
sql-resultset-nextrow 38
sql-resultset? 38
sql-rsmd 38
sql-rsmd-column-labels 38
sql-rsmd-column-names 38
sql-rsmd-column-typenames 38
sql-rsmd? 38
sql-statement 37
sql-statement-close 37
sql-statement-results 38
sql-statement? 37
string-append 7
string-ref 27
string? 18

Table of Contents

1 Introduction 1

1.1 What is BRL and Why Should I Care? 1
1.2 About This Manual 2
1.2.1 Further Help 2

2 Downloading and Installing BRL 3

2.1 Easy Installation via learnbrl.war 3
2.2 Prerequisites 3
2.3 Downloading BRL 4
2.4 Precompiled JAR Installation 4
2.5 Building from Source 5
2.6 Testing Your Installation 5

3 Introducing Scheme 7

3.1 Simple Expressions 7
3.2 Defining Procedures 8
3.3 Data Structures 9
3.4 Learning More 11

4 Learning BRL by Example 13

4.1 Web Page for Bruce 13
4.1.1 Understanding BRL Inside-Out 14
4.1.2 Bruce's Ruse Gets Trickier 15
4.2 Shared variables and common headers/footers 16
4.3 HTML Forms and CGI Environment Variables 16
4.4 Path Segments 19
4.5 Cookies 20
4.6 Session variables 20
4.7 Sending e-mail 21
4.8 SQL in BRL 22
4.8.1 One-Time Preparation for All Pages 22
4.8.2 Queries in an Individual Page 23
4.8.3 Inserts, Updates and Deletes 24
4.8.4 Grouping Results 26
4.8.5 URL and HTML Escapes 27
4.8.6 Searches and Sortable Columns 28
4.9 String Manipulation 30
4.9.1 Numbers to Strings 30
4.9.2 Dates to Strings 31
4.9.3 Trimming, Splitting and Joining 31
4.10 Connection Pools and Other Java Objects 32

5 BRL Reference 33

5.1 String Functions 33
5.2 Web Functions 35
5.3 SQL Functions 37
5.4 Network Functions 39
5.5 Miscellaneous Functions 40

6 Regular Expressions 43

6.1 Regexp Introduction 43
6.2 Regexp procedures provided 43
6.2.1 pregexp 44
6.2.2 pregexp-match-positions 44
6.2.3 pregexp-match 44
6.2.4 pregexp-replace 45
6.2.5 pregexp-replace* 45
6.3 The regexp pattern language 45
6.3.1 Basic assertions 45
6.3.2 Characters and character classes 46
6.3.2.1 Some frequently used character classes . . 46
6.3.2.2 POSIX character classes 46
6.3.3 Quantifiers 48
6.3.3.1 Numeric quantifiers 48
6.3.3.2 Non-greedy quantifiers 49
6.3.4 Clusters 49
6.3.4.1 Backreferences 50
6.3.4.2 Non-capturing clusters 51
6.3.4.3 Cloisters 51
6.3.5 Alternation 52
6.3.6 Backtracking 53
6.3.6.1 Disabling backtracking 53
6.3.7 Looking ahead and behind 53
6.3.7.1 Lookahead 54
6.3.7.2 Lookbehind 54
6.4 An extended example 54

Appendix A The Whys of BRL 57

A.1 Why not HTML-like syntax? 57
A.2 Why Scheme? 57
A.3 Why]string[? 58

Index 61