15

Domain specific languages

The combination of first class environments, lexical scoping, non-standard evaluation, and metaprogramming gives us a powerful toolkit for creating embedded domain specific languages (DSLs) in R. Embedded DSLs take advantage of a host language's parsing and execution framework, but adjust the semantics to make them more suitable for a specific task. DSLs are a very large topic, and this chapter will only scratch the surface, focussing on important implementation techniques rather than on how you might come up with the language in the first place. If you're interested in learning more, I highly recommend *Domain Specific Languages* (http://amzn.com/0321712943?tag=devtools-20) by Martin Fowler. It discusses many options for creating a DSL and provides many examples of different languages.

R's most popular DSL is the formula specification, which provides a succinct way of describing the relationship between predictors and the response in a model. Other examples include ggplot2 (for visualisation) and plyr (for data manipulation). Another package that makes extensive use of these ideas is dplyr, which provides `translate_sql()` to convert R expressions into SQL:

```
library(dplyr)
translate_sql(sin(x) + tan(y))
#> <SQL> SIN("x") + TAN("y")
translate_sql(x < 5 & !(y >= 5))
#> <SQL> "x" < 5.0 AND NOT(("y" >= 5.0))
translate_sql(first %like% "Had*")
#> <SQL> "first" LIKE 'Had*'
translate_sql(first %in% c("John", "Roger", "Robert"))
#> <SQL> "first" IN ('John', 'Roger', 'Robert')
translate_sql(like == 7)
#> <SQL> "like" = 7.0
```

This chapter will develop two simple, but useful DSLs: one to generate

© 2015 by Taylor & Francis Group, LLC

HTML, and the other to turn mathematical expressions expressed in R code into LaTeX.

Prerequisites

This chapter together pulls together many techniques discussed elsewhere in the book. In particular, you'll need to understand environments, functionals, non-standard evaluation, and metaprogramming.

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

15.1 HTML

HTML (hypertext markup language) is the language that underlies the majority of the web. It's a special case of SGML (standard generalised markup language), and it's similar but not identical to XML (extensible markup language). HTML looks like this:

```
<body>
  <h1 id='first'>A heading</h1>
  <p>Some text &amp; <b>some bold text.</b></p>
  <img src='myimg.png' width='100' height='100' />
</body>
```

Even if you've never looked at HTML before, you can still see that the key component of its coding structure is tags, `<tag></tag>`. Tags can be contained inside other tags and intermingled with text. Generally, HTML ignores whitespaces (a sequence of whitespace is equivalent to a single space) so you could put the previous example on a single line and it would still display the same in a browser:

```
<body><h1 id='first'>A heading</h1><p>Some text &amp; <b>some bold
text.</b></p><img src='myimg.png' width='100' height='100' />
</body>
```

However, like R code, you usually want to indent HTML to make the structure more obvious.

There are over 100 HTML tags. But to illustrate HTML, we're going to focus on just a few:

© 2015 by Taylor & Francis Group, LLC

- `<body>`: the top-level tag that all content is enclosed within
- `<h1>`: creates a heading-1, the top level heading
- `<p>`: creates a paragraph
- `<b>`: emboldens text
- `<img>`: embeds an image

(You probably guessed what these did already!)

Tags can also have named attributes. They look like `<tag a="a" b="b"></tag>`. Tag values should always be enclosed in either single or double quotes. Two important attributes used with just about every tag are `id` and `class`. These are used in conjunction with CSS (cascading style sheets) in order to control the style of the document.

Some tags, like `<img>`, can't have any content. These are called **void tags** and have a slightly different syntax. Instead of writing `<img></img>`, you write `<img />`. Since they have no content, attributes are more important. In fact, `img` has three that are used for almost every image: `src` (where the image lives), `width`, and `height`.

Because `<` and `>` have special meanings in HTML, you can't write them directly. Instead you have to use the HTML escapes: `&gt;` and `&lt;`. And, since those escapes use `&`, if you want a literal ampersand you have to escape with `&amp;`.

15.1.1 Goal

Our goal is to make it easy to generate HTML from R. To give a concrete example, we want to generate the following HTML with code that looks as similar to the HTML as possible.

```
<body>
  <h1 id='first'>A heading</h1>
  <p>Some text &amp; <b>some bold text.</b></p>
  <img src='myimg.png' width='100' height='100' />
</body>
```

To do so, we will work our way up to the following DSL:

```
with_html(body(
  h1("A heading", id = "first"),
  p("Some text &", b("some bold text.")),
  img(src = "myimg.png", width = 100, height = 100)
))
```

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

© 2015 by Taylor & Francis Group, LLC

Note that the nesting of function calls is the same as the nesting of tags: unnamed arguments become the content of the tag, and named arguments become their attributes. Because tags and text are clearly distinct in this API, we can automatically escape `&` and other special characters.

15.1.2 Escaping

Escaping is so fundamental to DSLs that it'll be our first topic. To create a way of escaping characters, we need to give "&" a special meaning without ending up double-escaping. The easiest way to do this is to create an S3 class that distinguishes between regular text (that needs escaping) and HTML (that doesn't).

```
html <- function(x) structure(x, class = "html")
print.html <- function(x, ...) {
  out <- paste0("<HTML> ", x)
  cat(paste(strwrap(out), collapse = "\n"), "\n", sep = "")
}
```

We then write an escape method that leaves HTML unchanged and escapes the special characters (`&`, `<`, `>`) for ordinary text. We also add a list method for convenience.

```
escape <- function(x) UseMethod("escape")
escape.html <- function(x) x
escape.character <- function(x) {
  x <- gsub("&", "&amp;", x)
  x <- gsub("<", "&lt;", x)
  x <- gsub(">", "&gt;", x)

  html(x)
}
escape.list <- function(x) {
  lapply(x, escape)
}

# Now we check that it works
escape("This is some text.")
#> <HTML> This is some text.
escape("x > 1 & y < 2")
```

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

© 2015 by Taylor & Francis Group, LLC

```
#> <HTML> x &gt; 1 &amp; y &lt; 2

# Double escaping is not a problem
escape(escape("This is some text. 1 > 2"))
#> <HTML> This is some text. 1 &gt; 2

# And text we know is HTML doesn't get escaped.
escape(html("<hr />"))
#> <HTML> <hr />
```

Escaping is an important component for many DSLs.

15.1.3 Basic tag functions

Next, we'll write a few simple tag functions and then figure out how to generalise this function to cover all possible HTML tags. Let's start with `<p>`. HTML tags can have both attributes (e.g., id or class) and children (like `<b>` or `<i>`). We need some way of separating these in the function call. Given that attributes are named values and children don't have names, it seems natural to separate using named arguments from unnamed ones. For example, a call to `p()` might look like:

```
p("Some text.", b("some bold text"), class = "mypara")
```

We could list all the possible attributes of the `<p>` tag in the function definition. However, that's hard not only because there are many attributes, but also because it's possible to use custom attributes (http://html5doctor.com/html5-custom-data-attributes/). Instead, we'll just use `...` and separate the components based on whether or not they are named. To do this correctly, we need to be aware of an inconsistency in `names()`:

```
names(c(a = 1, b = 2))
#> [1] "a" "b"
names(c(a = 1, 2))
#> [1] "a" ""
names(c(1, 2))
#> NULL
```

With this in mind, we create two helper functions to extract the named and unnamed components of a vector:

© 2015 by Taylor & Francis Group, LLC

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

```
named <- function(x) {
  if (is.null(names(x))) return(NULL)
  x[names(x) != ""]
}
unnamed <- function(x) {
  if (is.null(names(x))) return(x)
  x[names(x) == ""]
}
```

We can now create our `p()` function. Notice that there's one new function here: `html_attributes()`. It uses a list of name-value pairs to create the correct specification of HTML attributes. It's a little complicated (in part, because it deals with some idiosyncracies of HTML that I haven't mentioned.). However, because it's not that important and doesn't introduce any new ideas, I won't discuss it here (you can find the source online).

```
source("dsl-html-attributes.r", local = TRUE)
p <- function(...) {
  args <- list(...)
  attribs <- html_attributes(named(args))
  children <- unlist(escape(unnamed(args)))

  html(paste0(
    "<p", attribs, ">",
    paste(children, collapse = ""),
    "</p>"
  ))
}

p("Some text")
#> <HTML> <p>Some text</p>
p("Some text", id = "myid")
#> <HTML> <p id = 'myid'>Some text</p>
p("Some text", image = NULL)
#> <HTML> <p image>Some text</p>
p("Some text", class = "important", "data-value" = 10)
#> <HTML> <p class = 'important' data-value = '10'>Some
#> text</p>
```

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

© 2015 by Taylor & Francis Group, LLC

15.1.4 Tag functions

With this definition of `p()`, it's pretty easy to see how we can apply this approach to different tags: we just need to replace `"p"` with a variable. We'll use a closure to make it easy to generate a tag function given a tag name:

```
tag <- function(tag) {
  force(tag)
  function(...) {
    args <- list(...)
    attribs <- html_attributes(named(args))
    children <- unlist(escape(unnamed(args)))

    html(paste0(
      "<", tag, attribs, ">",
      paste(children, collapse = ""),
      "</", tag, ">"
    ))
  }
}
```

(We're forcing the evaluation of `tag` with the expectation that we'll be calling this function from a loop. This will help to avoid potential bugs caused by lazy evaluation.)

Now we can run our earlier example:

```
p <- tag("p")
b <- tag("b")
i <- tag("i")
p("Some text.", b("Some bold text"), i("Some italic text"),
  class = "mypara")
#> <HTML> <p class = 'mypara'>Some text.<b>Some bold
#> text</b><i>Some italic text</i></p>
```

Before we continue writing functions for every possible HTML tag, we need to create a variant of `tag()` for void tags. It can be very similar to `tag()`, but if there are any unnamed tags, it needs to throw an error. Also note that the tag itself will look slightly different:

```
void_tag <- function(tag) {
  force(tag)
```

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

© 2015 by Taylor & Francis Group, LLC

```
  function(...) {
    args <- list(...)
    if (length(unnamed(args)) > 0) {
      stop("Tag ", tag, " can not have children", call. = FALSE)
    }
    attribs <- html_attributes(named(args))

    html(paste0("<", tag, attribs, " />"))
  }
}

img <- void_tag("img")
img(src = "myimage.png", width = 100, height = 100)
#> <HTML> <img src = 'myimage.png' width = '100' height =
#> '100' />
```

15.1.5 Processing all tags

Next we need a list of all the HTML tags:

```
tags <- c("a", "abbr", "address", "article", "aside", "audio",
  "b","bdi", "bdo", "blockquote", "body", "button", "canvas",
  "caption","cite", "code", "colgroup", "data", "datalist",
  "dd", "del","details", "dfn", "div", "dl", "dt", "em",
  "eventsource","fieldset", "figcaption", "figure", "footer",
  "form", "h1", "h2", "h3", "h4", "h5", "h6", "head", "header",
  "hgroup", "html", "i","iframe", "ins", "kbd", "label",
  "legend", "li", "mark", "map","menu", "meter", "nav",
  "noscript", "object", "ol", "optgroup", "option", "output",
  "p", "pre", "progress", "q", "ruby", "rp","rt", "s", "samp",
  "script", "section", "select", "small", "span", "strong",
  "style", "sub", "summary", "sup", "table", "tbody", "td",
  "textarea", "tfoot", "th", "thead", "time", "title", "tr",
  "u", "ul", "var", "video")

void_tags <- c("area", "base", "br", "col", "command", "embed",
  "hr", "img", "input", "keygen", "link", "meta", "param",
  "source", "track", "wbr")
```

If you look at this list carefully, you'll see there are quite a few tags that have the same name as base R functions (`body`, `col`, `q`, `source`, `sub`, `summary`, `table`), and others that have the same name as popular packages

© 2015 by Taylor & Francis Group, LLC

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

(e.g., `map`). This means we don't want to make all the functions available by default, in either the global environment or the package environment. Instead, we'll put them in a list and add some additional code to make it easy to use them when desired. First, we make a named list:

```
tag_fs <- c(
  setNames(lapply(tags, tag), tags),
  setNames(lapply(void_tags, void_tag), void_tags)
)
```

This gives us an explicit (but verbose) way to call tag functions:

```
tag_fs$p("Some text.", tag_fs$b("Some bold text"),
  tag_fs$i("Some italic text"))
#> <HTML> <p>Some text.<b>Some bold text</b><i>Some
#> italic text</i></p>
```

We can then finish off our HTML DSL with a function that allows us to evaluate code in the context of that list:

```
with_html <- function(code) {
  eval(substitute(code), tag_fs)
}
```

This gives us a succinct API which allows us to write HTML when we need it but doesn't clutter up the namespace when we don't.

```
with_html(body(
  h1("A heading", id = "first"),
  p("Some text &", b("some bold text.")),
  img(src = "myimg.png", width = 100, height = 100)
))
#> <HTML> <body><h1 id = 'first'>A heading</h1><p>Some
#> text &amp;<b>some bold text.</b></p><img src =
#> 'myimg.png' width = '100' height = '100' /></body>
```

If you want to access the R function overridden by an HTML tag with the same name inside `with_html()`, you can use the full `package::function` specification.

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

© 2015 by Taylor & Francis Group, LLC

15.1.6 Exercises

1. The escaping rules for `<script>` and `<style>` tags are different: you don't want to escape angle brackets or ampersands, but you do want to escape `</script>` or `</style>`. Adapt the code above to follow these rules.
2. The use of `...` for all functions has some big downsides. There's no input validation and there will be little information in the documentation or autocomplete about how they are used in the function. Create a new function that, when given a named list of tags and their
attribute names (like below), creates functions which address this problem.

   ```
   list(
     a = c("href"),
     img = c("src", "width", "height")
   )
   ```

   All tags should get `class` and `id` attributes.
3. Currently the HTML doesn't look terribly pretty, and it's hard to see the structure. How could you adapt `tag()` to do indenting and formatting?

15.2 LaTeX

The next DSL will convert R expressions into their LaTeX math equivalents. (This is a bit like `?plotmath`, but for text instead of plots.) LaTeX is the lingua franca of mathematicians and statisticians: whenever you want to describe an equation in text (e.g., in an email), you write it as a LaTeX equation. Since many reports are produced using both R and LaTeX, it might be useful to be able to automatically convert mathematical expressions from one language to the other.

Because we need to convert both functions and names, this mathematical DSL will be more complicated than the HTML DSL. We'll also need to create a "default" conversion, so that functions we don't know about get a standard conversion. Like the HTML DSL, we'll also write functionals to make it easier to generate the translators.

Before we begin, let's quickly cover how formulas are expressed in LaTeX.

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

© 2015 by Taylor & Francis Group, LLC

15.2.1 LaTeX mathematics

LaTeX mathematics are complex. Fortunately, they are well documented (http://en.wikibooks.org/wiki/LaTeX/Mathematics). That said, they have a fairly simple structure:

- Most simple mathematical equations are written in the same way you'd type them in R: `x * y`, `z ^ 5`. Subscripts are written using `_` (e.g., `x_1`).
- Special characters start with a `\`: `\pi` = ÏĂ, `\pm` = Âś, and so on. There are a huge number of symbols available in LaTeX. Googling for `latex math symbols` will return many lists (http://www.sunilpatel.co.uk/latex-type/latex-math-symbols/). There's even a service (http://detexify.kirelabs.org/classify.html) that will look up the symbol you sketch in the browser.
- More complicated functions look like `\name{arg1}{arg2}`. For example, to write a fraction you'd use `\frac{a}{b}`. To write a square root, you'd use `\sqrt{a}`.
- To group elements together use `{}`: i.e., `x ^ a + b` vs. `x ^ {a + b}`.
- In good math typesetting, a distinction is made between variables and functions. But without extra information, LaTeX doesn't know whether `f(a * b)` represents calling the function `f` with input `a * b`, or is shorthand for `f * (a * b)`. If `f` is a function, you can tell LaTeX to typeset it using an upright font with `\textrm{f}(a * b)`.

15.2.2 Goal

Our goal is to use these rules to automatically convert an R expression to its appropriate LaTeX representation. We'll tackle this in four stages:

- Convert known symbols: `pi` -> `\pi`
- Leave other symbols unchanged: `x` -> `x`, `y` -> `y`
- Convert known functions to their special forms: `sqrt(frac(a, b))` -> `\sqrt{\frac{a, b}}`
- Wrap unknown functions with `\textrm`: `f(a)` -> `\textrm{f}(a)`

We'll code this translation in the opposite direction of what we did with the HTML DSL. We'll start with infrastructure, because that makes it easy to experiment with our DSL, and then work our way back down to generate the desired output.

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

© 2015 by Taylor & Francis Group, LLC

15.2.3 `to_math`

To begin, we need a wrapper function that will convert R expressions into LaTeX math expressions. This will work the same way as `to_html()`: capture the unevaluated expression and evaluate it in a special environment. However, the special environment is no longer fixed. It will vary depending on the expression. We do this in order to be able to deal with symbols and functions that we haven't yet seen.

```
to_math <- function(x) {
  expr <- substitute(x)
  eval(expr, latex_env(expr))
}
```

15.2.4 Known symbols

Our first step is to create an environment that will convert the special LaTeX symbols used for Greek, e.g., `pi` to `\pi`. This is the same basic trick used in `subset` that makes it possible to select column ranges by name (`subset(mtcars, , cyl:wt)`): bind a name to a string in a special environment.

We create that environment by naming a vector, converting the vector into a list, and converting the list into an environment.

```
greek <- c(
  "alpha", "theta", "tau", "beta", "vartheta", "pi", "upsilon",
  "gamma", "gamma", "varpi", "phi", "delta", "kappa", "rho",
  "varphi", "epsilon", "lambda", "varrho", "chi", "varepsilon",
  "mu", "sigma", "psi", "zeta", "nu", "varsigma", "omega", "eta",
  "xi", "Gamma", "Lambda", "Sigma", "Psi", "Delta", "Xi",
  "Upsilon", "Omega", "Theta", "Pi", "Phi")
greek_list <- setNames(paste0("\\", greek), greek)
greek_env <- list2env(as.list(greek_list), parent = emptyenv())
```

We can then check it:

```
latex_env <- function(expr) {
  greek_env
}

to_math(pi)
```

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

© 2015 by Taylor & Francis Group, LLC

```
#> [1] "\\pi"
to_math(beta)
#> [1] "\\beta"
```

15.2.5 Unknown symbols

If a symbol isn't Greek, we want to leave it as is. This is tricky because we don't know in advance what symbols will be used, and we can't possibly generate them all. So we'll use a little bit of metaprogramming to find out what symbols are present in an expression. The `all_names` function takes an expression and does the following: if it's a name, it converts it to a string; if it's a call, it recurses down through its arguments.

```
all_names <- function(x) {
  if (is.atomic(x)) {
    character()
  } else if (is.name(x)) {
    as.character(x)
  } else if (is.call(x) || is.pairlist(x)) {
    children <- lapply(x[-1], all_names)
    unique(unlist(children))
  } else {
    stop("Don't know how to handle type ", typeof(x),
      call. = FALSE)
  }
}

all_names(quote(x + y + f(a, b, c, 10)))
#> [1] "x" "y" "a" "b" "c"
```

We now want to take that list of symbols, and convert it to an environment so that each symbol is mapped to its corresponding string representation (e.g., so `eval(quote(x), env)` yields `"x"`). We again use the pattern of converting a named character vector to a list, then converting the list to an environment.

```
latex_env <- function(expr) {
  names <- all_names(expr)
  symbol_list <- setNames(as.list(names), names)
  symbol_env <- list2env(symbol_list)
```

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

© 2015 by Taylor & Francis Group, LLC

```
  symbol_env
}

to_math(x)
#> [1] "x"
to_math(longvariablename)
#> [1] "longvariablename"
to_math(pi)
#> [1] "pi"
```

This works, but we need to combine it with the Greek symbols environment. Since we want to give preference to Greek over defaults (e.g., `to_math(pi)` should give `"\\pi"`, not `"pi"`), `symbol_env` needs to be the parent of `greek_env`. To do that, we need to make a copy of `greek_env` with a new parent. While R doesn't come with a function for cloning environments, we can easily create one by combining two existing functions:

```
clone_env <- function(env, parent = parent.env(env)) {
  list2env(as.list(env), parent = parent)
}
```

This gives us a function that can convert both known (Greek) and unknown symbols.

```
latex_env <- function(expr) {
  # Unknown symbols
  names <- all_names(expr)
  symbol_list <- setNames(as.list(names), names)
  symbol_env <- list2env(symbol_list)

  # Known symbols
  clone_env(greek_env, symbol_env)
}

to_math(x)
#> [1] "x"
to_math(longvariablename)
#> [1] "longvariablename"
to_math(pi)
#> [1] "\\pi"
```

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

© 2015 by Taylor & Francis Group, LLC

15.2.6 Known functions

Next we'll add functions to our DSL. We'll start with a couple of helper closures that make it easy to add new unary and binary operators. These functions are very simple: they only assemble strings. (Again we use `force()` to make sure the arguments are evaluated at the right time.)

```
unary_op <- function(left, right) {
  force(left)
  force(right)
  function(e1) {
    paste0(left, e1, right)
  }
}

binary_op <- function(sep) {
  force(sep)
  function(e1, e2) {
    paste0(e1, sep, e2)
  }
}
```

Using these helpers, we can map a few illustrative examples of converting R to LaTeX. Note that with R's lexical scoping rules helping us, we can easily provide new meanings for standard functions like `+`, `-`, and `*`, and even `(` and `{`.

```
# Binary operators
f_env <- new.env(parent = emptyenv())
f_env$"+" <- binary_op(" + ")
f_env$"-" <- binary_op(" - ")
f_env$"*" <- binary_op(" * ")
f_env$"/" <- binary_op(" / ")
f_env$"^" <- binary_op("^")
f_env$"[" <- binary_op("_")

# Grouping
f_env$"{" <- unary_op("\\left{ ", " \\right}")
f_env$"(" <- unary_op("\\left( ", " \\right)")
f_env$paste <- paste

# Other math functions
```

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

© 2015 by Taylor & Francis Group, LLC

```
f_env$sqrt <- unary_op("\\sqrt{", "}")
f_env$sin <- unary_op("\\sin(", ")")
f_env$log <- unary_op("\\log(", ")")
f_env$abs <- unary_op("\\left| ", "\\right| ")
f_env$frac <- function(a, b) {
  paste0("\\frac{", a, "}{", b, "}")
}

# Labelling
f_env$hat <- unary_op("\\hat{", "}")
f_env$tilde <- unary_op("\\tilde{", "}")
```

We again modify `latex_env()` to include this environment. It should be the last environment R looks for names in: in other words, `sin(sin)` should work.

```
latex_env <- function(expr) {
  # Known functions
  f_env

  # Default symbols
  names <- all_names(expr)
  symbol_list <- setNames(as.list(names), names)
  symbol_env <- list2env(symbol_list, parent = f_env)

  # Known symbols
  greek_env <- clone_env(greek_env, parent = symbol_env)
}

to_math(sin(x + pi))
#> [1] "\\sin(x + \\pi)"
to_math(log(x_i ^ 2))
#> [1] "\\log(x_i^2)"
to_math(sin(sin))
#> [1] "\\sin(sin)"
```

15.2.7 Unknown functions

Finally, we'll add a default for functions that we don't yet know about. Like the unknown names, we can't know in advance what these will be, so we again use a little metaprogramming to figure them out:

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

© 2015 by Taylor & Francis Group, LLC

```
all_calls <- function(x) {
  if (is.atomic(x) || is.name(x)) {
    character()
  } else if (is.call(x)) {
    fname <- as.character(x[[1]])
    children <- lapply(x[-1], all_calls)
    unique(c(fname, unlist(children)))
  } else if (is.pairlist(x)) {
    unique(unlist(lapply(x[-1], all_calls), use.names = FALSE))
  } else {
    stop("Don't know how to handle type ", typeof(x), call. = FALSE)
  }
}

all_calls(quote(f(g + b, c, d(a))))
#> [1] "f" "+" "d"
```

And we need a closure that will generate the functions for each unknown call.

```
unknown_op <- function(op) {
  force(op)
  function(...) {
    contents <- paste(..., collapse = ", ")
    paste0("\\mathrm{", op, "}(", contents, ")")
  }
}
```

And again we update `latex_env()`:

```
latex_env <- function(expr) {
  calls <- all_calls(expr)
  call_list <- setNames(lapply(calls, unknown_op), calls)
  call_env <- list2env(call_list)

  # Known functions
  f_env <- clone_env(f_env, call_env)

  # Default symbols
  symbols <- all_names(expr)
  symbol_list <- setNames(as.list(symbols), symbols)
  symbol_env <- list2env(symbol_list, parent = f_env)
```

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

© 2015 by Taylor & Francis Group, LLC

```
  # Known symbols
  greek_env <- clone_env(greek_env, parent = symbol_env)
}

to_math(f(a * b))
#> [1] "\\mathrm{f}(a * b)"
```

15.2.8 Exercises

1. Add escaping. The special symbols that should be escaped by adding a backslash in front of them are `\`, `$`, and `%`. Just as with HTML, you'll need to make sure you don't end up double-escaping. So you'll need to create a small S3 class and then use that in function operators. That will also allow you to embed arbitrary LaTeX if needed.
2. Complete the DSL to support all the functions that `plotmath` supports.
3. There's a repeating pattern in `latex_env()`: we take a character vector, do something to each piece, convert it to a list, and then convert the list to an environment. Write a function that automates this task, and then rewrite `latex_env()`.
4. Study the source code for `dplyr`. An important part of its structure is `partial_eval()` which helps manage expressions when some of the components refer to variables in the database while others refer to local R objects. Note that you could use very similar ideas if you needed to translate small R expressions into other languages, like JavaScript or Python.

Downloaded by [The University of British Columbia Library] at 10:46 13 July 2016

© 2015 by Taylor & Francis Group, LLC