PL: Lecture #21

Tuesday, November 19th

Designing Domain Specific Languages (DSLs)

> 👉 ***PLAI §35***

Programming languages differ in numerous ways:

1. Each uses different notations for writing down programs. As we've observed, however, syntax is only partially interesting. (This is, however, less true of languages that are trying to mirror the notation of a particular domain.)
2. Control constructs: for instance, early languages didn't even support recursion, while most modern languages still don't have continuations.
3. The kinds of data they support. Indeed, sophisticated languages like Racket blur the distinction between control and data by making fragments of control into data values (such as first-class functions and continuations).
4. The means of organizing programs: do they have functions, modules, classes, namespaces, …?
5. Automation such as memory management, run-time safety checks, and so on.

Each of these items suggests natural questions to ask when you design your own languages in particular domains.

Obviously, there are a lot of domain specific languages these days — and that's not new. For example, four of the oldest languages were conceived as domain specific languages:

- **Fortran** — *Formula Translator*
- **Algol** — *Algorithmic Language*
- **Cobol** — *Common Business Oriented Language*
- **Lisp** — *List Processing*

Only in the late 60s / early 70s languages began to get free from their special purpose domain and become *general purpose* languages (GPLs). These days, we usually use some GPL for our programs and often come up with small *domain specific* languages (DSLs) for specific jobs. The problem is designing such a specific language. There are lots of decisions to make, and as should be clear now, many ways of shooting your self in the foot. You need to know:

- What is your domain?
- What are the common notations in this domain (need to be convenient both for the machine and for humans)?
- What do you expect to get from your DSL? (eg, performance gains when you know that you're dealing with a certain limited kind of functionality like arithmetics.)
- Do you have any semantic reason for a new language? (For example, using special scoping rules, or a mixture of lazy and eager evaluation, maybe a completely different way of evaluation (eg, makefiles).)
- Is your language expected to envelope other functionality (eg, shell scripts, TCL), perhaps throwing some functionality on a different language (makefiles and shell scripts), or is it going to be embedded in a bigger application (eg, PHP), or embedded in a way that exposes parts of an application to user automation (Emacs Lisp, Word Basic, Visual Basic for Office Application or Some Other Long List of Buzzwords).
- If you have one language embedded in another enveloping language — how do you handle syntax? How can they communicate (eg, share variables)?

And very important:

- Is there a benefit for implementing a DSL over using a GPL — how much will your DSL grow (usually more than you think)? Will it get to a point where it will need the power of a full GPL? Do you want to risk doing this just to end up admitting that you need a “Real Language” and dump your solution for “Visual Basic for Applications”? (It might be useful to think ahead about things that you know you don’t need, rather than things you need.)

To clarify why this can be applicable in more situations than you think, consider what programming languages are used for. One example that should not be ignored is using a programming language to implement a programming language — for example, what we did so far (or any other interpreter or compiler). In the same way that some piece of code in a PL represent functions about the “real world”, there are other programs that represent things in a language — possibly even the same one. To make a side-effect-full example, the meaning of `one-brick` might abstract over laying a brick when making a wall — it abstracts all the little details into a function:

```
(define (one-brick wall brick-pile)
  (move-eye (location brick-pile))
  (let ([pos (find-available-brick-position brick-pile)])
    (move-hand pos)
    (grab-object))
  (move-eye wall)
  (let ([pos (find-next-brick-position wall)])
    (move-hand pos)
    (drop-object)))
```

and we can now write

```
(one-brick my-wall my-brick-pile)
```

instead of all of the above. We might use that in a loop:

```
(define (build-wall wall pile)
  (define (loop n)
    (when (< n 500)
      (one-brick wall pile)
      (loop (add1 n))))
  (loop 0))
```

This is a common piece of looping code that we’ve seen in many forms, and a common complaint of newcomers to functional languages is the lack of some kind of a loop. But once you know the template, writing such loops is easy — and in fact, you can write code that would take something like:

```
(define (build-wall wall pile)
  (loop-for i from 1 to 500
    (one-brick wall pile)))
```

and produce the previous code. Note the main point here: we switch from code that deals with bricks to code that deals with code.

Now, a viable option for implementing a new DSL is to do so by transforming it into an existing language. Such a process is usually tedious and error prone — tedious because you need to deal with the boring parts of a language (making a parser etc), and error prone because it’s easy to generate bad code (especially when you’re dealing with strings) and you get bad errors in terms of the translated code instead of the actual code, resorting to debugging the intermediate generated programs. Lisp languages traditionally have taken this idea one level further than other languages: instead of writing a new transformer for your language, you use the host language, but you extend and customize it by adding you own forms.

Syntax Transformations: Macros

> 👉 ***PLAI §36***

Macros are one of the biggest advantages of all Lisps, and specifically even more so an advantage of Scheme implementations, and yet more specifically, it is a major Racket feature: this section is therefore specific to Racket (which has this unique feature), although most of this is the same in most Schemes.

As we have previously seen, it is possible to implement one language construct using another. What we did could be described as bits of a compiler, since they translate one language to another.

We will see how this can be done *in* Racket: implementing some new linguistic forms in terms of ones that are already known. In essence, we will be translating Racket constructs to other Racket constructs — and all that is done *in Racket*, no need to go back to the language Racket was implemented in (C).

This is possible with a simple "trick": the Racket implementation uses some syntax objects. These objects are implemented somehow inside Racket's own source code. But these objects are also directly available for our use — part of the implementation is exposed to our level. This is quite similar to the way we have implemented pairs in our language — a TOY or a SLOTH pair is implemented using a Racket pair, so the *same* data object is available at both levels.

This is the big idea in Lisp, which Scheme (and therefore Racket) inherited from (to some extent): programs are made of numbers, strings, symbols and lists of these — and these are all used both at the meta-level as well as the user level. This means that instead of having no meta-language at all (locking away a lot of useful stuff), and instead of having some crippled half-baked meta language (CPP being both the most obvious (as well as the most popular) example for such a meta language), instead of all this we get exactly the same language at both levels.

How is this used? Well, the principle is simple. For example, say we want to write a macro that will evaluate two forms in sequence, but if the first one returns a result that is not false then it returns it instead of evaluating the second one too. This is exactly how `or` behaves, so pretend we don't have it — call our version `orelse`:

```
(orelse <expr1> <expr2>)
```

in effect, we add a new *special* form to our language, with its own evaluation rule, only we know how to express this evaluation rule by translating it to things that are already part of our language.

We could do this as a simple function — only if we're willing to explicitly delay the arguments with a `lambda`, and use the thunks in the function:

```
(define (orelse thunk1 thunk2)
  (if (thunk1)
    (thunk1) ; ignore the double evaluation for now
    (thunk2)))
```

or:

```
(define (orelse thunk1 thunk2)
  ((if (thunk1)
     thunk1
     thunk2)))
```

and using it like this:

```
(orelse (lambda () 1) (lambda () (error "boom")))
```

But this is clearly not the right way to do this: whoever uses this code must be aware of the need to send us thunks, and it's verbose and inconvenient.

> *Note that this could be a feasible solution if there was a uniform way to have an easy syntactic way to say "a chunk of code" instead of immediately execute it — this is exactly what `(lambda () ...)` does. So we could, for example, make `{...}` be a shorthand for that, which is what Perl-6 is doing. However, we will soon see examples where we want more than just delay the evaluation of some code.*

We want to translate

```
(orelse <expr1> <expr2>)

--to-->

(if <expr1>
  <expr1>
  <expr2>)
```

If we look at the code as an s-expression, then we can write the following function:

```
(define (translate-orelse l)
  (if (and (list? l)
           (= 3 (length l))
           (eq? 'orelse (first l)))
    (list 'if (second l) (second l) (third l))
    (error 'translate-orelse "bad input: ~s" l)))
```

We can now try it with a simple list:

```
(translate-orelse '(orelse foo1 foo2))
```

and note that the result is correct.

How is this used? Well, all we need is to hook *our* function into our implementation's evaluator. In Lisp, we get a `defmacro` form for this, and many Schemes inherited it or something similar. In Racket, we need to

```
(require compatibility/defmacro)
```

but it requires the transformation to be a little different in a way that makes life easier: the above contains a lot of boilerplate code. Usually, we will require the input to be a list of some known length, the first element to be a symbol that specifies our form, and then do something with the other arguments. So we'd want to always follow a template that looks like:

```
(define (translate-??? exprs)
  (if (and (list? exprs)
           (= N (length exprs))
           (eq? '??? (car exprs)))
    (let ([E1 (cadr exprs)]
          [E2 (caddr exprs)]
          ...)
      ...make result expression...)
    (error ...)))
```

But this looks very similar to making sure that a function call is a specific function call (and for a good reason — macro usages look just like function calls). So make the translation function get a number of arguments one each for each part of the input, an s-expression. For example, the above translation and test become:

```
(define (translate-orelse <expr1> <expr2>)
  (list 'if <expr1> <expr1> <expr2>))

(translate-orelse 'foo1 'foo2)
```

The number of arguments is used to check the input (turning an arity error for the macro to an arity error for the translator function call), and we don't need to "caddr our way" to arguments.

This gives us the simple definition — but what about the promised hook? — All we need is to use `define-macro` instead of `define`, and change the name to the name that will trigger this translation (providing the last missing test of the input):

```
(define-macro (orelse <expr1> <expr2>)
  (list 'if <expr1> <expr1> <expr2>))
```

and test it:

```
(orelse 1 (error "boom"))
```

Note that this is basically a (usually purely functional) lazy language of transformations which is built on top of Racket. It is possible for macros to generate pieces of code that contain references to these same macros, and they will be used to expand those instances again.

Now we start with the problems, one by one.

Macro Problems

There is an inherent problem when macros are being used, in any form and any language (even in CPP): you must remember that you are playing with *expressions*, not with values — which is why this is problematic:

```
(define (foo x) (printf "foo ~s!\n" x) x)

(or (foo 1) (foo 2))

(orelse (foo 1) (foo 2))
```

And the reason for this should be clear. The standard solution for this is to save the value as a binding — so back to the drawing board, we want this transformation instead:

```
(orelse <expr1> <expr2>)
-->
(let ((val <expr1>))
  (if val
    val
    <expr2>))
```

(Note that we would have the same problem in the version that used simple functions and thunks.)

And to write the new code:

```
(define-macro (orelse <expr1> <expr2>)
  (list 'let (list (list 'val <expr1>))
    (list 'if 'val
      'val
      <expr2>)))

(orelse (foo 1) (foo 2))
```

and this works like we want it to.

Complexity of S-expression transformations

As can be seen, writing a simple macro doesn't look too good — what if we want to write a more complicated macro? A solution to this is to look at the above macro and realize that it *almost* looks like the code we want — we basically want to return a list of a certain fixed shape, we just want some parts to be filled in by the given arguments. Something like:

```
(define-macro (orelse <expr1> <expr2>)
  '(let ((val <expr1>))
     (if val
       val
       <expr2>)))
```

if we had a way to make the `<...>`s not be a fixed part of the result, but we actually want the values that the transformation function received. (Remember that the `<` and `>` are just a part of the name, no magic, just something to make these names stand out.) This is related to notational problems that logicians and philosophers had problems with for centuries. One solution that Lisp uses for this is: instead of a quote, use backquote (called `quasiquote` in Racket) which is almost like quote, except that you can `unquote` parts of the value inside. This is done with a "`,`" comma. Using this, the above code can be written like this:

```
(define-macro (orelse <expr1> <expr2>)
  `(let ((val ,<expr1>))
     (if val
       val
       ,<expr2>)))
```

Scoping problems

You should be able to guess what's this problem about. The basic problem of these macros is that they cannot be used reliably — the name that is produced by the macro can shadow a name that is in a completely different place, therefore destroying lexical scope. For example, in:

```
(let ((val 4))
  (orelse #f val))
```

the `val` in the macro shadows the use of this name in the above. One way to solve this is to write macros that look like this:

```
(define-macro (orelse <expr1> <expr2>)
  `(let ((%%!my*internal*var-do-not-use!%% ,<expr1>))
     (if %%!my*internal*var-do-not-use!%%
       %%!my*internal*var-do-not-use!%%
       ,<expr2>)))
```

or:

```
(define-macro (orelse <expr1> <expr2>)
  `(let ((i-am-using-orelse-so-i-should-not-use-this-name ,<expr1>))
     (if i-am-using-orelse-so-i-should-not-use-this-name
       i-am-using-orelse-so-i-should-not-use-this-name
       ,<expr2>)))
```

or (this is actually similar to using UUIDs):

```
(define-macro (orelse <expr1> <expr2>)
  `(let ((eli@barzilay.org/foo/bar/2002-02-02-10:22:22 ,<expr1>))
     (if eli@barzilay.org/foo/bar/2002-02-02-10:22:22
       eli@barzilay.org/foo/bar/2002-02-02-10:22:22
       ,<expr2>)))
```

Which is really not too good because such obscure variables tend to clobber each other too, in all kinds of unexpected ways.

Another way is to have a function that gives you a different variable name every time you call it:

```
(define-macro (orelse <expr1> <expr2>)
  (let ((temp (gensym)))
    `(let ((,temp ,<expr1>))
       (if ,temp
         ,temp
         ,<expr2>))))
```

but this is not safe either since there might still be clashes of these names (eg, if they're using a counter that is specific to the current process, and you start a new process and load code that was generated earlier). The Lisp solution for this (which Racket's `gensym` function implements as well) is to use *uninterned* symbols — symbols that have their own identity, much like strings, and even if two have the same name, they are not `eq?`.

Note also that there is the mirror side of this problem — what happens if we try this:

```
(let ([if 123]) (orelse #f #f))
```

? This leads to capture in the other direction — the code above shadows the `if` binding that the macro produces.

Some Schemes will allow something like

```
(define-macro (foo x)
  `(,mul-list ,x))
```

but this is a hack since the macro outputs something that is not a pure s-expression (and it cannot work for a syntactic keyword like `if`). Specifically, it is not possible to write the resulting expression (to a compiled file, for example).

We will ignore this for a moment.

Another problem — manageability of these transformations.

Quasiquotes gets us a long way, but it is still insufficient.

For example, lets write a Racket `bind` that uses `lambda` for binding. The transformation we now want is:

```
(bind ((var expr) ...)
  body)
-->
((lambda (var ...) body)
 expr ...)
```

The code for this looks like this:

```
(define-macro (bind var-expr-list body)
  (cons (list 'lambda (map car var-expr-list) body)
        (map cadr var-expr-list)))
```

This already has a lot more pitfalls. There are `list`s and `cons`es that you should be careful of, there are `map`s and there are `cadr`s that would be catastrophic if you use `car`s instead. The quasiquote syntax is a little more capable — you can write this:

```
(define-macro (bind var-expr-list body)
  `((lambda ,(map car var-expr-list) ,body)
    ,@(map cadr var-expr-list)))
```

where "`,@`" is similar to "`,`" but the unquoted expression should evaluate to a list that is *spliced* into its surrounding list (that is, its own parens are removed and it's made into elements in the containing list). But this is still not as readable as the transformation you actually want, and worse, it is not checking that the input syntax is valid, which can lead to very confusing errors.

This is yet another problem — if there is an error in the resulting syntax, the error will be reported in terms of this result rather than the syntax of the code. There is no easy way to tell where these errors are coming from. For example, say that we make a common mistake: forget the "`@`" character in the above macro:

```
(define-macro (bind var-expr-list body)
  `((lambda ,(map car var-expr-list) ,body)
    ,(map cadr var-expr-list)))
```

Now, someone else (the client of this macro), tries to use it:

```
> (bind ((x 1) (y 2)) (+ x y))
procedure application: expected procedure,
given: 1; arguments were: 2
```

Yes? Now what? Debugging this is difficult, since in most cases it is not even clear that you were using a macro, and in any case the macro comes from code that you have no knowledge of and no control over. [The problem in this specific case is that the macro expands the code to:

```
((lambda (x y) (+ x y))
 (1 2))
```

so Racket will to use `1` as a function and throw a runtime error.]

Adding error checking to the macro results in this code:

```
(define-macro (bind var-expr-list body)
  (if (andmap (lambda (var-expr)
                (and (list? var-expr)
                     (= 2 (length var-expr))
                     (symbol? (car var-expr))))
              var-expr-list)
    `((lambda ,(map car var-expr-list) ,body)
      ,@(map cadr var-expr-list))
    (error 'bind "bad syntax whaaaa!")))
```

Such checks are very important, yet writing this is extremely tedious.

Scheme (and Racket) Macros

Scheme, Racket included (and much extended), has a solution that is better than `defmacro`: it has `define-syntax` and `syntax-rules`. First of all, `define-syntax` is used to create the “magical connection” between user code and some macro transformation code that does some rewriting. This definition:

```
(define-syntax foo
  ...something...)
```

makes `foo` be a special syntax that, when used in the head of an expression, will lead to transforming the expression itself, where the result of this transformation is what gets used instead of the original expression. The “`...something...`” in this code fragment should be a transformation function — one that consumes the expression that is to be transformed, and returns the new expression to run.

Next, `syntax-rules` is used to create such a transformation in an easy way. The idea is that what we thought to be an informal specification of such rewrites, for example:

- `let` can be defined as the following transformation:

  ```
  (let ((x v) ...) body ...)
  --> ((lambda (x ...) body ...) v ...)
  ```

- `let*` is defined with *two* transformation rules:

  1. (let* () body …) –> (let () body …)
  2. (let* ((x1 v1) (x2 v2) …) body …) –> (let ((x1 v1)) (let* ((x2 v2) …) body …))

can actually be formalized by automatically creating a syntax transformation function from these rule specifications. (Note that this example has round parentheses so we don’t fall into the illusion that square brackets are different: the resulting transformation would be the same.) The main point is to view the left hand side as a *pattern* that can match some forms of syntax, and the right hand side as producing an output that can use some matched patterns.

`syntax-rules` is used with such rewrite specifications, and it produces the corresponding transformation function. For example, this:

```
(syntax-rules () ;*** ignore this "()" for now
  [(x y) (y x)])
```

evaluates to a function that is somewhat similar to:

```
(lambda (expr)
  (if (and (list? expr) (= 2 (length expr)))
    (list (second expr) (first expr))
    (error "bad syntax")))
```

but `match` is a little closer, since it uses similar input patterns:

```
(lambda (expr)
  (match expr
    [(list x y) (list y x)]
    [else (error "bad syntax")]))
```

Such transformations are used in a `define-syntax` expression to tie the transformer back into the compiler by hooking it on a specific keyword. You can now appreciate how all this work when you see how easy it is to define macros that are very tedious with `define-macro`. For example, the above `bind`:

```
(define-syntax bind
  (syntax-rules ()
    [(bind ((x v) ...) body ...)
     ((lambda (x ...) body ...) v ...)]))
```

and `let*` with its two rules:

```
(define-syntax let*
  (syntax-rules ()
    [(let* () body ...)
     (let () body ...)]
    [(let* ((x v) (xs vs) ...) body ...)
     (let ((x v)) (let* ((xs vs) ...) body ...))]))
```

These transformations are so convenient to follow, that Scheme specifications (and reference manuals) describe forms by specifying their definition. For example, the Scheme report, specifies `let*` as a “derived form”, and explains its semantics via this transformation.

The input patterns in these rules are similar to `match` patterns, and the output patterns assemble an s-expression using the matched parts in the input. For example:

```
(x y) --> (y x)
```

does the thing you expect it to do — matches a parenthesized form with two sub-forms, and produces a form with the two sub-forms swapped. The rules for “`...`” on the left side are similar to `match`, as we have seen many times, and on the right side it is used to *splice* a matched sequence into the resulting expression and it is required to use the `...` for sequence-matched pattern variables. For example, here is a list of some patterns, and a description of how they match an input when used on the left side of a transformation rule and how they produce an output expression when they appear on the right side:

- `(x ...)`

  > ***LHS:*** *matches a parenthesized sequence of zero or more expressions, and the `x` pattern variable is bound to this whole sequence; `match` analogy: `(match ? [(list x ...) ?])`*
  >
  > ***RHS:*** *when `x` is bound to a sequence, this will produce a parenthesized expression containing this sequence; `match` analogy: `(match ? [(list x ...) x])`*

- `(x1 x2 ...)`

> ***LHS:*** *matches a parenthesized sequence of one or more expressions, the first is bound to* `x1` *and the rest of the sequence is bound to* `x2`*;*
>
> `match` *analogy:* `(match ? [(list x1 x2 ...) ?])`
>
> ***RHS:*** *produces a parenthesized expression that contains the expression bound to* `x1` *first, then all of the expressions in the sequence that* `x2` *is bound to;*
>
> `match` *analogy:* `(match ? [(list x1 x2 ...) (cons x1 x2)])`

- `((x y) ...)`

> ***LHS:*** *matches a parenthesized sequence of 2-form parenthesized sequences, binding* `x` *to all the first forms of these, and* `y` *to all the seconds of these (so they will both have the same number of items);*
>
> `match` *analogy:* `(match ? [(list (list x y) ...) ?])`
>
> ***RHS:*** *produces a list of forms where each one is made of consecutive forms in the* `x` *sequence and consecutive forms in the* `y` *sequence (both sequences should have the same number of elements);*
>
> `match` *analogy:*
>
> ```
> (match ? [(list (list x y) ...)
>          (map (lambda (x y) (list x y)) x y)])
> ```

Some examples of transformations that would be very tedious to write code manually for:

- `((x y) ...) --> ((y x) ...)`

  Matches a sequence of 2-item sequences, produces a similar sequence with all of the nested 2-item sequences flipped.

- `((x y) ...) --> ((x ...) (y ...))`

  Matches a similar sequence, and produces a sequence of two sequences, one of all the first items, and one of the second ones.

- `((x y ...) ...) --> ((y ... x) ...)`

  Similar to the first example, but the nested sequences can have 1 or more items in them, and the nested sequences in the result have the first element moved to the end. Note how the `...` are nested: the rule is that for each pattern variable you count how many `...`s apply to it, and that tells you what it holds — and you have to use the same `...` nestedness for it in the output template.

This is solving the problems of easy code — no need for `list`, `cons` etc, not even for quasiquotes and tedious syntax massaging. But there were other problems. First, there was a problem of bad scope, one that was previously solved with a `gensym`:

```
(define-macro (orelse <expr1> <expr2>)
  (let ((temp (gensym)))
    `(let ((,temp ,<expr1>))
       (if ,temp ,temp ,<expr2>))))
```

Translating this to `define-syntax` and `syntax-rules` we get something simpler:

```
(define-syntax orelse
  (syntax-rules ()
    [(orelse <expr1> <expr2>)
     (let ((temp <expr1>))
       (if temp temp <expr2>))]))
```

Even simpler, Racket has a macro called `define-syntax-rule` that expands to a `define-syntax` combined with a `syntax-rules` — using it, we can write:

```
(define-syntax-rule (orelse <expr1> <expr2>)
  (let ((temp <expr1>))
    (if temp temp <expr2>)))
```

This looks like like a function — but you must remember that it is a transformation rule specification which is a *very* different beast, as we'll see.

The main thing here is that Racket takes care of making bindings follow the lexical scope rules:

```
(let ([temp 4])
  (orelse #f temp))
```

works fine. In fact, it fully respects the scoping rules: there is no confusion between bindings that the macro introduces and bindings that are introduced where the macro is used. (Think about different colors for bindings introduced by the macro and other bindings.) It's also fine with many cases that are much harder to cope with otherwise (eg, cases where there is no `gensym` magic solution):

```
(let ([if +])
  (orelse 1 1))

(let ([if +])
  (if (orelse 1 1) 10 100)) ; two different `if's here
```

or combining both:

```
(let ([if #f] [temp 4])
  (orelse if temp))
```

(You can try DrRacket's macro debugger to see how the various bindings get colored differently.)

`define-macro` advocates will claim that it is difficult to make a macro that intentionally plants a *known* identifier. Think about a `loop` macro that has an `abort` that can be used inside its body. Or an `if-it` form that is like `if`, but makes it possible to use the condition's value in the "then" branch as an `it` binding. It is possible with all Scheme macro systems to "break hygiene" in such ways, and we will later see how to do this in Racket. However, Racket also provides a better way to deal with such problems (think about `it` being always "bound to a syntax error", but locally rebound in an `if-it` form).

Scheme macros are said to be *hygienic* — a term used to specify that they respect lexical scope. There are several implementations of hygienic macro systems across Scheme implementations, Racket uses the one known as "syntax-case system", named after the `syntax-case` construct that we discuss below.

> *All of this can get much more important in the presence of a module system, since you can write a module that provides transformations rules, not just values and functions. This means that the concept of "a library" in Racket is something that doesn't exist in other languages: it's a library that has values, functions, as well as macros (or, "compiler plugins").*

The way that Scheme implementations achieve hygiene in a macro system is by making it deal with more than just raw S-expressions. Roughly speaking, it deals with syntax objects that are sort of a wrapper structure around S-expression, carrying additional information. The important part of this information when it gets to dealing with hygiene is the "lexical scope" — which can roughly be described as having identifiers be represented as symbols plus a "color" which represents the scope. This way such systems can properly avoid confusing identifiers with the same name that come from different scopes.

There was also the problem of making debugging difficult, because a macro can introduce errors that are "coming out of nowhere". In the implementation that we work with, this is solved by adding yet more information to these syntax objects — in addition to the underlying S-expression and the lexical scope, they also contain source location information. This allows Racket (and DrRacket) to locate the source of a specific syntax error, so locating the offending code is easy. DrRacket's macro debugger heavily relies on this information to provide a very useful tool — since writing macros can easily become a hard job.

Finally, there was the problem of writing bad macros. For example, it is easy to forget that you're dealing with a macro definition and write:

```
(define-syntax-rule (twice x) (+ x x))
```

just because you want to inline the addition — but in this case you end up duplicating the input expression which can have a disastrous effect. For example:

```
(twice (twice (twice (twice (twice (twice (twice (twice 1)))))))))
```

expands to a *lot* of code to compile.

Another example is:

```
(define-syntax-rule (with-increment var expr)
  (let ([var (add1 var)]) expr))
...
(with-increment (* foo 2)
  ...code...)
```

the problem here is that (* foo 2) will be used as an identifier to be bound by the `let` expression — which can lead to a confusing syntax error.

Racket provides many tools to help macro programmers — in addition to a user-interface tool like the macro debugger there are also programmer-level tools where you can reject an input if it doesn't contain an identifier at a certain place etc. Still, writing macros is much harder than writing functions — some of these problems are inherent to the problem that macros solve; for example, you may *want* a `twice` macro that replicates an expression. By specifying a transformation to the core language, a macro writer has full control over which expressions get evaluated and how, which identifiers are binding instances, and how is the scope of the given expression is shaped.

Meta Macros

One of the nice results of `syntax-rules` dealing with the subtle points of identifiers and scope is that things works fine even when we "go up a level". For example, the short `define-syntax-rule` form that we've seen is *itself* a defined as a simple macro:

```
(define-syntax define-syntax-rule
  (syntax-rules ()
    [(define-syntax-rule (name P ...) B)
     (define-syntax name
       (syntax-rules ()
         [(name P ...) B]))]))
```

In fact, this is very similar to something that we have already seen: the `rewrite` form that we have used in Schlac is implemented in just this way. The only difference is that `rewrite` requires an actual `=>` token to separate the input pattern from the output template. If we just use it in a syntax rule:

```
(define-syntax rewrite
  (syntax-rules ()
    [(rewrite (name P ...) => B)
     (define-syntax name
       (syntax-rules ()
         [(name P ...) B]))]))
```

it won't work. Racket treats the above `=>` just like any identifier, which in this case acts as a pattern variable which matches anything. The solution to this is to list the `=>` as a keyword which is expected to appear in the macro use as-is — and that's what the mysterious `()` of `syntax-rules` is used for: any identifier listed there is taken to be such a keyword. This makes the following version

```
(define-syntax rewrite
  (syntax-rules (=>)
    [(rewrite (name P ...) => B)
     (define-syntax name
       (syntax-rules ()
         [(name P ...) B]))]))
```

do what we want and throw a syntax error unless `rewrite` is used with an actual `=>` in the proper place.