PL: Lecture #23

Tuesday, November 19th

Solving the `syntax-rules` problems extra

So far it looks like we didn't do anything new, but the important change is already in: the fact that the results of a macro is a plain Racket expression mean that we can now add more API functionality for dealing with syntax values. There is no longer a problem with running "meta-level" code vs generated runtime code: anything that is inside a `syntax` (anything that is quoted with a "`#'`") is generated code, and the rest is code that is executed when the macro expands. We will now introduce some of the Racket macro API by demonstrating the solutions to the `syntax-rules` problem that were mentioned earlier.

First of all, we've talked about the problem of reporting good errors. For example, make this:

```
(for 1 = 1 to 3 do ...)
```

throw a proper error instead of leaving it for `lambda` to complain about. To make it easier to play with, we'll use a simpler macro:

```
(define-syntax fun
  (syntax-rules (->)
    [(_ id -> E) (lambda (id) E)])) ; _ matches the head `fun'
```

and using an explicit function:

```
(define-syntax (fun stx)
  (syntax-case stx (->)
    [(_ id -> E) #'(lambda (id) E)]))
```

One of the basic API functions is `syntax-e` — it takes in a syntax value and returns the S-expression that it wraps. In this case, we can pull out the identifier from this, and check that it is a valid identifier using `symbol?` on what it wraps:

```
(define-syntax (fun stx)
  (syntax-case stx (->)
    [(_ id -> E)
     (if (symbol? (syntax-e (cadr (syntax-e stx))))
       #'(lambda (id) E)
       (error 'fun "bad syntax: expecting an identifier, got ~s"
                   (cadr (syntax-e stx))))]))
```

The error is awkward though — it doesn't look like the usual kind of syntax errors that Racket throws: it's shown in an ugly way, and its source is not properly highlighted. A better way to do this is to use `raise-syntax-error' — it takes an error message, the offending syntax, and the offending part of this syntax:

```
(define-syntax (fun stx)
  (syntax-case stx (->)
    [(_ id -> E)
     (if (symbol? (syntax-e (cadr (syntax-e stx))))
       #'(lambda (id) E)
       (raise-syntax-error
        'fun "bad syntax: expecting an identifier"
        stx (cadr (syntax-e stx))))]))
```

Another inconvenient issue is with pulling out the identifier. Consider that `#'(lambda (id) E)` is a new piece of syntax that has the supposed identifier in it — we pull it from that instead of from `stx`, but it would be even easier with `#'(id)`, and even easier than that with just `#'id` which will be just the identifier:

```
(define-syntax (fun stx)
  (syntax-case stx (->)
    [(_ id -> E)
     (if (symbol? (syntax-e #'id))
       #'(lambda (id) E)
       (raise-syntax-error
        'fun "bad syntax: expecting an identifier"
        stx #'id))]))
```

Also, checking that something is an identifier is common enough that there is another predicate for this (the combination of `syntax-e` and `symbol?`) — `identifier?`:

```
(define-syntax (fun stx)
  (syntax-case stx (->)
    [(_ id -> E)
     (if (identifier? #'id)
       #'(lambda (id) E)
       (raise-syntax-error
        'fun "bad syntax: expecting an identifier"
        stx #'id))]))
```

As a side note, checking the input pattern for validity is very common, and in some cases might be needed to discriminate patterns (eg, one result when `id` is an identifier, another when it's not). For this, `syntax-cases` clauses have "guard expressions" — so we can write the above more simply as:

```
(define-syntax (fun stx)
  (syntax-case stx (->)
    [(_ id -> E)
     (identifier? #'id)
     #'(lambda (id) E)]))
```

This, however, produces a less informative "bad syntax" error, since there is no way to tell what the error message should be. (There is a relatively new Racket tool called `syntax-parse` where such requirements can be specified and a proper error message is generated on bad inputs.)

We can now resolve the `repeat` problem — create a `(repeat N E)` macro:

```
(define-syntax (repeat stx)
  (define (n-copies n expr)
    (if (> n 0) (cons expr (n-copies (sub1 n) expr)) null))
  (syntax-case stx ()
    [(_ N E)
     (integer? (syntax-e #'N))
     #'(begin (n-copies (syntax-e #'N) #'E))]))
```

(Note that we can define an internal helper function, just like we do with plain functions.) But this doesn't quite work (and if you try it, you'll see an interesting error message) — the problem is that we're *generating* code with a call to `n-copies` in it, instead of actually calling it. The problem is that we need to take the list that `n-copies` generates, and somehow "plant" it in the resulting syntax. So far the only things that were planted in it are pattern variables — and we can actually use another `syntax-case` to do just that: match the result of `n-copies` against a pattern variable, and then use that variable in the final syntax:

```
(define-syntax (repeat stx)
  (define (n-copies n expr)
    (if (> n 0) (cons expr (n-copies (sub1 n) expr)) null))
  (syntax-case stx ()
    [(_ N E)
     (number? (syntax-e #'N))
```

```
        (syntax-case (n-copies (syntax-e #'N) #'E) ()
          [(expr ...) #'(begin expr ...)])]))
```

This works — but one thing to note here is that `n-copies` returns a list, not a syntax. The thing is that `syntax-case` will automatically "coerce" S-expressions into a syntax in some way, easy to do in this case since we only care about the elements of the list, and those are all syntaxes.

However, this use of `syntax-case` as a pattern variable binder is rather indirect, enough that it's hard to read the code. Since this is a common use case, there is a shorthand for that too: `with-syntax`. It looks as a kind of a `let`-like form, but instead of binding plain identifiers, it binds pattern identifiers — and in fact, the things to be bound are themselves patterns:

```
(define-syntax (repeat stx)
  (define (n-copies n expr)
    (if (> n 0) (cons expr (n-copies (sub1 n) expr)) null))
  (syntax-case stx ()
    [(_ N E)
     (number? (syntax-e #'N))
     (with-syntax ([(expr ...) (n-copies (syntax-e #'N) #'E)])
       #'(begin expr ...))]))
```

Note that there is no need to implement `with-syntax` as a primitive form — it is not too hard to implement it as a macro that expands to the actual use of `syntax-case`. (In fact, you can probably guess now that the Racket core language is much smaller than it seems, with large parts that are implemented as layers of macros.)

There is one more related group of shorthands that is relevant here: `quasisyntax`, `unsyntax`, and `unsyntax-splicing`. These are analogous to the quoting forms by the same names, and they have similar shorthands: "`#``", "`#,`" and "`#,@`". They could be used to implement this macro:

```
(define-syntax (repeat stx)
  (define (n-copies n expr)
    (if (> n 0) (cons expr (n-copies (sub1 n) expr)) null))
  (syntax-case stx ()
    [(_ N E)
     (number? (syntax-e #'N))
     #`(begin #,@(n-copies (syntax-e #'N) #'E))]))
```

[As you might suspect now, these new forms are also implemented as macros, which expand to the corresponding uses of `with-syntax`, which in turn expand into `syntax-case` forms.]

We now have almost enough machinery to implement the `rev-app` macro, and compare it to the original (complex) version that used `syntax-rules`. The only thing that is missing is a way to generate a number of new identifiers — which we achieved earlier by a number of macro expansion (each expansion of a macro that has a new identifier `x` will have this identifier different from other expansions, which is why it worked). Racket has a function for this: `generate-temporaries`. Since it is common to generate temporaries for input syntaxes, the function expects an input syntax that has a list as its S-expression form (or a plain list).

```
(define-syntax (rev-app stx)
  (syntax-case stx ()
    [(_ F E ...)
     (let ([temps (generate-temporaries #'(E ...))])
       (with-syntax ([(E* ...) (reverse (syntax-e #'(E ...)))]
                     [(x  ...) temps]
                     [(x* ...) (reverse temps)])
         #'(let ([x* E*] ...)
             (F x ...))))]))

;; see that it works
```

```
    (define (show x) (printf ">>> ~s\n" x) x)
    (rev-app list (show 1) (show 2) (show 3))
```

Note that this is not shorter than the `syntax-rules` version, but it is easier to read since `reverse` and `generate-temporaries` have an obvious direct intention, eliminating the need to wonder through rewrite rules and inferring how they do their work. In addition, this macro expands in one step (use the macro stepper to compare it with the previous version), which makes it much more efficient.

Breaking Hygiene, How Bad is it? extra

We finally get to address the second deficiency of `syntax-rules` — its inability to intentionally capture an identifier so it is visible in user code. Let's start with the simple version, the one that didn't work:

```
    (define-syntax-rule (if-it E1 E2 E3)
      (let ([it E1]) (if it E2 E3)))
```

and translate it to `syntax-case`:

```
    (define-syntax (if-it stx)
      (syntax-case stx ()
        [(if-it E1 E2 E3)
         #'(let ([it E1]) (if it E2 E3))]))
```

The only problem here is that the `it` identifier is introduced by the macro, or more specifically, by the `syntax` form that makes up the return syntax. What we need now is a programmatic way to create an identifier with a lexical context that is different than the default. As mentioned above, Racket's syntax system (and all other `syntax-case` systems) doesn't provide a direct way to manipulate the lexical context. Instead, it provides a way to create a piece of syntax by copying the lexical scope of another one — and this is done with the `datum->syntax` function. The function consumes a syntax value to get the lexical scope from, and a "datum" which is an S-expression that can contain syntax values. The result will have these syntax values as given on the input, but raw S-expressions will be converted to syntaxes, using the given lexical context. In the above case, we need to convert an `it` symbol into the same-named identifier, and we can do that using the lexical scope of the input syntax. As we've seen before, we use `with-syntax` to inject the new identifier into the result:

```
    (define-syntax (if-it stx)
      (syntax-case stx ()
        [(if-it E1 E2 E3)
         (with-syntax ([it (datum->syntax stx 'it)])
           #'(let ([it E1]) (if it E2 E3)))]))
```

We can even control the scope of the user binding — for example, it doesn't make much sense to have `it` in the `else` branch. We can do this by first binding a plain (hygienic) identifier to the result, and only bind `it` to that when needed:

```
    (define-syntax (if-it stx)
      (syntax-case stx ()
        [(if-it E1 E2 E3)
         (with-syntax ([it (datum->syntax stx 'it)])
           #'(let ([tmp E1]) (if tmp (let ([it tmp]) E2) E3)))]))
```

[A relevant note: Racket provides something that is known as "The Macro Writer's Bill of Rights" — in this case, it guarantees that the extra `let` does not imply a runtime or a memory overhead.]

This works — and it's a popular way for creating such user-visible bindings. However, breaking hygiene this way can lead to some confusing problems. Such problems are usually visible when we try to compose macros — for example, say that we want to create a `cond-it` macro, the anaphoric analogue of `cond`, which binds `it` in each branch. It seems that an obvious way of doing this is by layering it on top of `if-it` — it should even be simple enough to be defined with `syntax-rules`:

```
(define-syntax cond-it
  (syntax-rules (else)
    [(_ [test1 expr1 ...] [tests exprs ...] ...)
     (if-it test1
       (begin expr1 ...)
       (cond-it [tests exprs ...] ...))]
    ;; two end cases -- one with an `else' and one without
    [(_ [else expr ...]) (begin expr ...)]
    [(_) (void)]))
```

Surprisingly, this does not work! Can you see what went wrong?

The problem lies in how the `it` identifier is generated — it used the lexical context of the whole `if-it` expression, which seemed like exactly what we wanted. But in this case, the `if-it` expression is coming from the `cond-it` macro, not from the user input. Or to be more accurate: it's the `cond-it` macro which is the user of `if-it`, so `it` is visible to `cond-it`, but not to its own users…

Note that these anaphoric macros are a popular example, but these problems do pop up elsewhere too. For example, imagine a loop macro that wants to bind `break` unhygienically, a class macro that binds `this`, and many others.

How can we solve this? There are several ways for this:

- Don't break hygiene. For example, instead of `if-it` and `cond-it` forms that have an implicit `it`, use forms with an explicit identifiers. For example: `(if* it <test> <then> <else>)`. This might be a little more verbose at times, but it makes everything behave very well, since the identifiers always have the right scope.
- Try to patch things up with a little more unhygienic in your macros. In this case, try to make `cond-it` introduce `if-it` unhygienically, so when it introduces `it` in its own turn, it will be the right one. This is bad, since we started trying to get hygienic macros, and there are no easy discounts. (For example, what if there's a different `if-it` that is used at the place where `cond-it` is used?) In fact, the unhygienic `define-macro` that we've seen is an extreme example of this: there is no lexical scope anywhere; so `it` is the same identifier no matter where it's introduced. But as we've seen, this means that hygiene is always broken when possible.
- Try to make `cond-it` come up with its own unhygienic `it`, then bind this `it` to the `it` that `if-it` creates. This can work but on one hand it's difficult and fragile to write such code, and on the other hand it defeats the simplicity of macros.
- Finally, Racket provides an elegant solution in the form of *syntax parameters*. The idea is to avoid the unhygienic binding: have a single global binding for `it`, and change the meaning of this binding on uses of `if-it`. (If you're interested, see "Keeping it Clean with Syntax Parameters" for details.)

Macros in Racket's Module System extra

> *Not in PLAI*

One of the main things that Racket pioneered is integrating its syntax system with its module system. In plain Racket (`#lang racket`, not the course languages), every file is a module that can `provide` some functionality, for when you put this code in a file:

```
#lang racket
(provide plus)
(define (plus x y) (+ x y))
```

You get a library that gives you a `plus` function. This is just the usual thing that you'd expect from a library facility in a language — but Racket allows you to do the same with syntax definitions. For example, if we add the following to this file:

```
(provide with)
(define-syntax-rule (with [x V] E)
  (let ([x V]) E))
```

we — the users of this library — also get to have a `with` binding, which is a “FLANG-compatibility” macro that expands into a `let`. Now, on a brief look, this doesn’t seem all too impressive, but consider the fact that `with` is actually a translation function that lives at the syntax level, as a kind of a compiler plugin, and you’ll see that this is not as trivial as it seems. Racket arranges to do this with a concept of *instantiating* code at the compiler level, so libraries are used in two ways: either the usual thing as a runtime instantiation, or at compile time.

Defining Languages in Racket extra

But then Racket takes this concept even further. So far, we treated the thing that follows a `#lang` as a kind of a language specification — but the more complete story is that this specification is actually just a *module*. The only difference between such modules like `racket` or `pl` and “library modules” as our above file is that language modules provide a bunch of functionality that is specific to a language implementation. However, you don’t need to know about these things up front: instead, there’s a few tools that allow you to provide everything that some other module provides — if we add this to the above:

```
(provide (all-from-out racket))
```

then we get a library that provides the same two bindings as above (`plus` and `with`) — *in addition* to everything from the `racket` library (which it got from its own `#lang racket` line).

To use this file as a language, the last bit that we need to know about is the actual concrete level syntax. Racket provides an `s-exp` language which is a kind of a meta language for reading source code in the usual S-expression syntax. Assuming that the above is in a file called `mylang.rkt`, we can use it (from a different file in the same directory) as follows:

```
#lang s-exp "mylang.rkt"
```

which makes the language of this file be (a) read using the S-expression syntax, and (b) get its bindings from our module, so

```
#lang s-exp "mylang.rkt"
(with [x 10] (* x 4))
```

will show a result of `40`.

So far this seems like just some awkward way to get to the same functionality as a simple library — but now we can use more tools to make things more interesting. First, we can provide everything from `racket` *except* for `let` — change the last `provide` to:

```
(provide (except-out (all-from-out racket) let))
```

Next, we can provide our `with` but make it have the name `let` instead — by replacing that `(provide with)` with:

```
(provide (rename-out [with let]))
```

The result is a language that is the same as Racket, except that it has an additional `plus` “built-in” function, and its `let` syntax is different, as specified by our macro:

```
#lang s-exp "mylang.rkt"
(let [x 10] (plus x 4))
```

To top things off, there are a few “special” implicit macros that Racket uses. One of them, `#%app`, is a macro that is used implicitly whenever there’s an expression that looks like a function application. In our terms, that’s the `Call` AST node that gets used whenever a braced-form isn’t one of the known forms. If we override this macro in a similar way that we did for `let`, we're essentially changing the semantics of application syntax. For example, here's a definition that makes it possible to use a `@` keyword to get a list of results of applying a function on several arguments:

```
(define-syntax my-app
  (syntax-rules (@)
    [(_ F @ E ...)
     (list (F E) ...)]
    [(_ x ...) (x ...)]))
```

This makes the `(my-app add1 @ 1 2)` application evaluate to `'(2 3)`, but if `@` is not used (as the second subexpression), we get the usual function application. (Note that this is because the last clause expands to `(x ...)` which implicitly has the usual Racket function application.) We can now make our language replace Racket's implicit `#%app` macro with this, in the same way as we did before: first, drop Racket's version from what we `provide`:

```
(provide (except-out (all-from-out racket) let #%app))
```

and then `provide` our definition instead

```
(provide (rename-out [my-app #%app]))
```

Users of our language get this as the regular function application:

```
#lang s-exp "mylang.rkt"
(let [x (plus 6 10)] (sqrt @ (plus x -7) x (plus x 9)))
```

Since `#%app` is a macro, it can evaluate to anything, even to things that are not function applications at all. For example, here's an extended definition that adds an arrow syntax that expands to a `lambda` expression not to an actual application:

```
(define-syntax my-app
  (syntax-rules (@ =>)
    [(_ F @ E ...)
     (list (F E) ...)]
    [(_ x => E ...)
     (lambda x E ...)]
    [(_ x ...) (x ...)]))
```

And an example of using it

```
#lang s-exp "mylang.rkt"
(define add1 ((x) => (+ x 1)))
;; or, combining all application forms in one example:
(((x) => (plus x 7)) @ 10 20 30)
```

Another such special macro is `#%module-begin`: this is a macro that is wrapped around the whole module body. Changing it makes it possible to change the semantics of a sequence of toplevel expressions in our language. The following is our complete language, with an example of redefining `#%module-begin` to create a "verbose" language that prints out expressions and what they evaluate to (note the `verbose` helper macro that is completely internal):

```
;; A language that is built as an extension on top of Racket

#lang racket

(provide (except-out (all-from-out racket)
                     let #%app #%module-begin))

(provide plus)
(define (plus x y) (+ x y))
```

```
(provide (rename-out [with let]))
(define-syntax-rule (with [x V] E)
                    (let ([x V]) E))

(provide (rename-out [my-app #%app]))
(define-syntax my-app
  (syntax-rules (=> @)
    [(_ x => E ...)
     (lambda x E ...)]
    [(_ F @ E ...)
     (list (F E) ...)]
    [(_ x ...) (x ...)]))

(provide (rename-out [mod-beg #%module-begin]))
(define-syntax-rule (mod-beg E ...)
                    (#%module-begin (verbose E) ...))
(define-syntax verbose
  (syntax-rules ()
    [(_ (define name value)) ; assume no (define (foo ...) ...)
     (begin (define name value)
            (printf "~s := ~s\n" 'name name))]
    [(_ E)
     (printf "~s --> ~s\n" 'E E)]))
```

And for reference, try that language with the above example:

```
#lang s-exp "mylang.rkt"
(define seven (+ 3 4))
(define add1 ((x) => (+ x 1)))
(((x) => (plus x seven)) @ 10 20 30)
```

Macro Conclusions

☞ ***PLAI §37.5***

Macros are extremely powerful, but this also means that their usage should be restricted only to situations where they are really needed. You can view any function as extending the current collection of tools that you provide — where these tools are much more difficult for your users to swallow than plain functions: evaluation can happen in any way, with any scope, unlike the uniform rules of function application. An analogy is that every function (or value) that you provide is equivalent to adding nouns to a vocabulary, but macros can add completely new rules for reading, since using them might result in a completely different evaluation. Because of this, adding macros carelessly can make code harder to read and debug — and using them should be done in a way that is as clear as possible for users.

When should a macro be used?

- Providing cosmetics: eliminating some annoying repetitiveness and/or inconvenient verbosity. This is usually macros that are intended to beautify code, for example, we could use a macro to make this bit of the Sloth source:

  ```
  (list '+ (box (racket-func->prim-val + #t)))
  (list '- (box (racket-func->prim-val - #t)))
  (list '* (box (racket-func->prim-val + #t)))
  ```

  look much better, by using a macro instead of the above. We can try to use a function, but we still need two inputs for each call — the name and the function:

```
(rfpv '+ + #t)
(rfpv '- - #t)
(rfpv '* + #t)
```

and a macro can eliminate this (small, but potentially dangerous) redundancy. For example:

```
(define-syntax-rule (rfpv fun flag)
  (list 'fun (box (racket-func->prim-val fun flag))))
```

and then:

```
(rfpv + #t)
(rfpv - #t)
(rfpv * #t)
```

eliminates the typo that was in the previous examples (did you catch that?).

- Altering the order of evaluation: as seen with the `orelse` macro, we can control evaluation order in our macro. This is achieved by translating the macro into Racket code with a known evaluation order. We even choose not to evaluate some parts, or evaluate some parts multiple times (eg, the `for` macro).

  Note that by itself, we could get this if only we had a more light-weight notation for thunks, since then we could simply use functions. For example, a `while` function could easily be used with thunks:

  ```
  (define (while cond body)
    (when (cond)
      (body)
      (while cond body)))
  ```

  if the syntax for a thunk would be as easy as, for example, using curly braces:

  ```
  (let ([i 0])
    (while { (< i 10) }
      { (printf "i = ~s\n" i) (set! i (+ i 1)) }))
  ```

- Introducing binding constructs: macros that have a different binding structure from Racket built-ins. These kind of macros are ones that makes a powerful language — for example, we've seen how we can survive without basic built-ins like `let`. For example, the `for` macro has its own binding structure.

  Note that with a sufficiently concise syntax for functions such as the arrow functions in JavaScript, we can get away with plain functions here too. For example, instead of a `with` macro, we could do it with a function:

  ```
  function with(val,fun) { return fun(val); }
  with( 123, x => x*x );
  ```

  (The obvious inconvenience is that the order can be weird.)

- Defining data languages: macros can be used for expressions that are not Racket expressions themselves. For example, the parens that wrap binding pairs in a `let` form are not function applications. Some times it is possible to use quotes for that, but then we get run-time values rather than being able to translate them into Racket code. Another usage of this category is to hide representation details that might involve implicit lambda's (for example, `delay`) — if we define a macro, then there is a single point where we control whether an expression is used within some `lambda` — but it it was a function, we'd have to change every usage of it to add an explicit lambda.

It is also important to note that macros should not be used too frequently. As said above, every macro adds a completely different way of reading your code — a way that doesn't use the usual "nouns" and "verbs", but there are other reasons not to use a macro.

One common usage case is as an optimization — trying to avoid an extra function call. For example, this:

```
  int min(int x, int y) {
    if ( x < y ) then return x; else return y;
```

```
    }
```

might seem wasteful if you don’t want a full function call on every usage of `min`. So you might be tempted to use this instead:

```
    #define min(x,y) x<y ? x : y
```

you even know the pitfalls of C macros so you make it more robust:

```
    #define min(x,y) (((x)<(y)) ? (x) : (y))
```

But small functions like the above are things that any decent compiler should know how to optimize, and even if your compiler doesn’t, it’s still not worth doing this optimization because programmer time is the most expensive factor in any computer system. In addition, a compiler is committed to doing these optimizations only when possible (eg, it is not possible to in-line a recursive function) and to do proper in-lining — unlike the `min` CPP macro above which is erroneous in case `x` or `y` are expressions that have side-effects.

Side-note: macros in mainstream languages

Macros are an extremely powerful tool in Racket (and other languages in the Lisp family) — how come nobody else uses them?

Well, people have tried to use them in many contexts. The problem is that you cannot get away with a simple solution that does nothing more than textual manipulation of your programs. For example, the standard C preprocessor is a macro language, but it is fundamentally limited to very simple situations. This is still a hot topic these days, with modern languages trying out different solutions (or giving up and claiming that macros are evil).

Here is an example that was written by Mark Jason Dominus (“Higher Order Perl”), in a **Perl mailing list post** among further discussion on macros in Lisp vs other languages, including Perl’s source transformers that are supposed to fill a similar role.

The example starts with writing the following simple macro:

```
    #define square(x) x*x
```

This doesn’t quite work because

```
    2/square(10)
```

expands to

```
    2/10*10
```

which is 2, but you wanted 0.02. So you need this instead:

```
    #define square(x) (x*x)
```

but this breaks because

```
    square(1+1)
```

expands to

```
    (1+1*1+1)
```

which is 3, but you wanted 4. So you need this instead:

```
    #define square(x) ((x)*(x))
```

But what about

```
x = 2;
square(x++);
```

which expands to

```
((x++)*(x++))
```

or an expensive expression? So you need this instead:

```
int __MYTMP;
#define square(x) (__MYTMP = (x), __MYTMP*__MYTMP)
```

but now it only works for ints; you can't do square(3.5) any more. To really fix this you have to use nonstandard extensions, something like:

```
#define square(x) ({ typedef xtype = x; xtype xval = x; xval*xval; })
```

or more like:

```
#define square(x) \
  ({ typedef xtype = (x); \
     xtype xval = (x); \
     xval*xval; })
```

And that's just to get trivial macros, like "square()", to work.

You should be able to appreciate now the tremendous power of macros. This is why there are so many "primitive features" of programming languages that can be considered as merely library functionality given a good macro system. For example, people are used to think about OOP as some inherent property of a language — but in Racket there are at least two very different object systems that it comes with, and several others in user-distributed code. All of these are implemented as a library which provides the functionality as well as the necessary syntax in the form of macros. So the basic principle is to have a small core language with powerful constructs, and make it easy to express complex ideas using these constructs.

This is an important point to consider before starting a new DSL (reminder: domain specific language) — if you need something that looks like a simple DSL but might grow to a full language, you can extend an existing language with macros to have the features you want, and you will always be able to grow to use the full language if necessary. This is particularly easy with Racket, but possible in other languages too.

Side note: the principle of a powerful but simple code language and easy extensions is not limited to using macros — other factors are involved, like first-class functions. In fact, "first class"-ness can help in many situations, for example: single inheritance + classes as first-class values can be used instead of multiple inheritance.