13 Desugaring as a Language Feature

We have thus far extensively discussed and relied upon desugaring, but our current desguaring mechanism have been weak. We have actually used desugaring in two different ways. One, we have used it to *shrink* the language: to take a large language and distill it down to its core [REF]. But we have also used it to *grow* the language: to take an existing language and add new features to it [REF]. This just shows that desugaring is a tremendously useful feature to have. Indeed, it is so useful that we might ask two questions:

- Because we create languages to simplify the creation of common tasks, what would a language designed to support desugaring look like? Note that by "look" we don't mean only syntax but also its key behavioral properties.
- Given that general-purpose languages are often used as a target for desugaring, why don't they offer desugaring capabilities *in the language itself*? For instance, this might mean extending a base language with the additional language that is the response to the previous question.

We are going to explore the answer to both questions simultaneously, by studying the facilities provided for this by Racket.

13.1 A First Example

Remember that in [REF] we added `let` as syntactic sugar over `lambda`. The pattern we followed was this:

DrRacket has a very useful tool called the Macro Stepper, which shows the step-by-step expansion of programs. You should try all the examples in this chapter using the Macro Stepper. For now, however, you should run them in `#lang plai` rather than `#lang plai-typed`.

```
(let (var val) body)
```

is transformed into

```
((lambda (var) body) val)
```

Do Now!

If this doesn't sound familiar, now would be a good time to refresh your memory of why this works.

The simplest way of describing this transformation would be to state it directly: to write, somehow,

```
(let (var val) body)
->
((lambda (var) body) val)
```

In fact, this is almost precisely what Racket enables you to do.

We'll use the name `my-let` instead of `let` Because the latter is already defined in Racket.

```
(define-syntax my-let-1
  (syntax-rules ()
    [(my-let-1 (var val) body)
     ((lambda (var) body) val)]))
```

`syntax-rules` tells Racket that whenever it sees an expression with `my-let-1` immediately after the opening parenthesis, it should check that it follows the pattern `(my-let-1 (var val) body)`. The `var`, `val` and `body` are *syntactic variables*: they are variables that stand for bodies of code. In particular, they match whatever s-expression is in that position. If the expression matches the pattern, then the syntactic variables are bound to the corresponding parts of the expression, and become available for use in the right-hand side.

You may have noticed some additional syntax, such as `()`. We'll explain this later.

The right-hand side—in this case, `((lambda (var) body) val)`—is the output generated. Each of the syntactic variables are replaced with the corresponding parts of the input using our old friend, substitution. This substitution is utterly simplistic: it makes no attempt to. Thus, if we were to try using it with

```
(my-let-1 (3 4) 5)
```

Racket would not initially complain that `3` is provided in an identifier position; rather, it would let the identifier percolate through, desugaring this into

```
((lambda (3) 5) 4)
```

which in turn produces an error:

```
lambda: expected either <id> or ‘[<id> : <type>]’
  for function argument in: 3
```

This immediately tells us that the desugaring process is straightforward in its function: it doesn't attempt to guess or be clever, but instead simply rewrites while substituting. The output is an expression that is again subject to desugaring.

As a matter of terminology, this simple form of expression-rewriting is often called a *macro*, as we mentioned earlier in [REF]. Traditionally this form of desugaring is called macro *expansion*, though this term is misleading because the output of desugaring can be smaller than the input (though it is usually larger).

Of course, in Racket, a `let` can bind multiple identifiers, not just one. If we were to write this informally, say on a board, we might write something like `(let ([var val] ...) body) -> ((lambda (var ...) body) val ...)` with the `...` meaning "zero or more", and the intent being that the `var ...` in the output should correspond to the sequence of `var`s in the input. Again, this is almost precisely Racket syntax:

```
(define-syntax my-let-2
  (syntax-rules ()
    [(my-let-2 ([var val] ...) body)
     ((lambda (var ...) body) val ...)]))
```

Observe the power of the `...` notation: the sequence of "pairs" in the input is turned into a pair of sequences in the output; put differently, Racket "unzips" the input sequence. Conversely, this same notation can be used to zip together sequences.

13.2 Syntax Transformers as Functions

Earlier we saw that `my-let-1` does not even attempt to ensure that the syntax in the identifier position is truly (i.e., syntactically) an identifier. We cannot remedy that with the `syntax-rules` mechanism, but we can with a much more powerful mechanism called `syntax-case`. Because `syntax-case` exhibits many other useful features as well, we'll introduce it and then grow it gradually.

The first thing to understand is that a macro is actually a *function*. It is not, however, a function from regular run-time values to other run-time values, but rather a function *from syntax to syntax*. These functions execute in a world whose purpose is to *create the program to execute*. Observe that we're talking about the program *to* execute: the actual execution of the program may only occur much later (or never at all). This point is actually extremely clear when we examine desugaring, which is very explicitly a function from (one kind of) syntax to (another kind of) syntax. This is perhaps obscured above in two ways:

- The notation of `syntax-rules`, with no explicit parameter name or other "function header", may not make clear that it is a functional transformation (though the rewriting rule format does allude to this fact).
- In desugaring, we specify one atomic function for the entire process. Here, we are actually writing several little functions, one for each kind of new syntactic construct (such as `my-let-1`), and these pieces are woven together by an invisible function that controls the overall rewriting process. (As a concrete example, it is not inherently clear that the output of a macro is expanded further—though a simple example immediately demonstrates that this is indeed the case.)

Exercise

Write one or more macros to confirm that the output of a macro is expanded further.

There is one more subtlety. Because the form of a macro looks rather like Racket code, it is not immediately clear that it "lives in another world". In the abstract, it may be helpful to imagine that the macro definitions are actually written in an entirely different language that processes only syntax. This simplicity is, however, misleading. In practice, program transformers—also called *compilers*—are full-blown programs, too, and need all the power of ordinary programs. This would have necessitated the creation of a parallel language purely for processing programs. This would be wasteful and pointless; therefore, Racket instead endows syntax-transforming programs with the full power of Racket itself.

With that prelude, let's now introduce `syntax-case`. We'll begin by simply rewriting `my-let-1` (under the name `my-let-3`) using this new notation. First, we have to write a header for the definition; notice already the explicit parameter:

<***sc-macro-eg***> **::=**

```
(define-syntax (my-let-3 x)
  <sc-macro-eg-body>)
```

This binds `x` to the entire `(my-let-3 ...)` expression.

As you might imagine, `define-syntax` simply tells Racket you're about to define a new macro. It does not pick precisely how you want to implement it, leaving you free to use any mechanism that's convenient. Earlier we used `syntax-rules`; now we're going to use `syntax-case`. In particular, `syntax-case` needs to explicitly be given access to the expression to pattern-match:

<***sc-macro-eg-body***> **::=**

```
(syntax-case x ()
  <sc-macro-eg-rule>)
```

Now we're ready to express the rewrite we wanted. Previously a rewriting rule had two parts: the structure of the input and the corresponding output. The same holds here. The first (matching the input) is the same as before, but the second (the output) is a little different:

<***sc-macro-eg-rule***> **::=**

```
[(my-let-3 (var val) body)
 #'((lambda (var) body) val)]
```

Observe the crucial extra characters: `#'`. Let's examine what that means.

In `syntax-rules`, the entire output part simply specifies the structure of the output. In contrast, because `syntax-case` is laying bare the functional nature of transformation, the output part is in fact an arbitrary expression that may perform any computations it wishes. It must simply evaluate to a piece of syntax.

Syntax is actually a distinct datatype. As with any distinct dataype, it has its own rules for construction. Concretely, we construct syntax values by writing `#'`; the following s-expression is treated as a syntax value. (In case you were wondering, the `x` bound in the macro definition above is also of this datatype.)

The syntax constructor, `#'`, enjoys a special property. Inside the output part of the macro, all syntax variables in the input are automatically bound, and replaced on occurrence. As a result, when the expander encounters `var` in the output, say, it replaces `var` with the corresponding part of the input expression.

Do Now!

> Remove the `#'` and try using the above macro definition. What happens?

So far, `syntax-case` merely appears to be a more complicated form of `syntax-rules`: perhaps slightly better in that it more cleanly delineates the functional nature of expansion, and the type of output, but otherwise simply more unwieldy. As we will see, however, it also offers significant power.

Exercise

> `syntax-rules` can actually be expressed as a *macro* over `syntax-case`. Define it.

13.3 Guards

Now we can return to the problem that originally motivated the introduction of `syntax-case`: ensuring that the binding position of a `my-let-3` is syntactically an identifier. For this, you need to know one new feature of `syntax-case`: each rewriting rule can have two parts (as above), or three. If there are three present, the *middle* one is treated as a *guard*: a predicate that must evaluate to true for expansion to proceed rather than signal a syntax error. Especially useful in this context is the predicate `identifier?`, which determines whether a syntax object is syntactically an identifier (or variable).

Do Now!

> Write the guard and rewrite the rule to incorporate it.

Hopefully you stumbled on a subtlety: the argument to `identifier?` is of type *syntax*. It needs to refer to the actual fragment of syntax bound to `var`. Recall that `var` is bound in the syntax space, and `#'` substitutes identifiers bound there. Therefore, the correct way to write the guard is:

```
(identifier? #'var)
```

With this information, we can now write the entire rule:

<sc-macro-eg-guarded-rule> ::=

```
[(my-let-3 (var val) body)
 (identifier? #'var)
 #'((lambda (var) body) val)]
```

Do Now!

> Now that you have a guarded rule definition, try to use the macro with a non-identifier in the binding position and see what happens.

13.4 Or: A Simple Macro with Many Features

Consider `or`, which implements disjunction. It is natural, with prefix syntax, to allow `or` to have an arbitrary number of sub-terms. We expand `or` into nested conditionals that determine the truth of the expression.

13.4.1 A First Attempt

Let's try a first version of `or`:

```
(define-syntax (my-or-1 x)
  (syntax-case x ()
    [(my-or-1 e0 e1 ...)
     #'(if e0
           e0
           (my-or-1 e1 ...))]))
```

It says that we can provide any number of sub-terms (more on this in a moment). Expansion rewrites this into a conditional that tests the first sub-term; if this is a true value it returns that value (more on *this* in a moment!), otherwise it is the disjunction of the remaining terms.

Let's try this on a simple example. We would expect this to evaluate to `true`, but instead:

```
> (my-or-1 #f #t)
my-or-1: bad syntax in: (my-or-1)
```

What happened? This expression turned into

```
(if #f
    #f
    (my-or-1 #t))
```

which in turn expanded into

```
(if #f
    #f
    (if #t
        #t
        (my-or-1)))
```

for which there is no definition. That's because the pattern `e0 e1 ...` means *one or more* sub-terms, but we ignored the case when there are zero. What happens when there are no sub-terms? The identity for disjunction is falsehood.

Exercise

Why is `#f` the right default?

By filling it in below, we illustrate a macro that has more than one rule. Macro rules are matched sequentially, so we should be sure to put the most specific rules first, lest they get overridden by more general ones (though in this particular case, the two rules are non-overlapping). This yields our improved macro:

```
(define-syntax (my-or-2 x)
  (syntax-case x ()
    [(my-or-2)
     #'#f]
    [(my-or-2 e0 e1 ...)
     #'(if e0
           e0
           (my-or-2 e1 ...))]))
```

which now expands as we expect. In what follows, we will find it useful to have a special case when there is only a single sub-term:

```
(define-syntax (my-or-3 x)
  (syntax-case x ()
    [(my-or-3)
     #'#f]
    [(my-or-3 e)
     #'e]
    [(my-or-3 e0 e1 ...)
     #'(if e0
           e0
           (my-or-3 e1 ...))]))
```

This keeps the output of expansion more concise, which we will find useful below.

Observe that in this version of the macro, the patterns are *not* disjoint: the third (one-or-more sub-terms) subsumes the second (one sub-term). Therefore, it is essential that the second rule not swap with the third.

13.4.2 Guarding Evaluation

We said above that this expands as we expect. Or does it? Let's try the following example:

```
(let ([init #f])
  (my-or-3 (begin (set! init (not init))
                  init)
           #f))
```

Observe that `or` returns the actual value of the first "truthy" value, so the developer can use it in further computations. Therefore, this returns the value of `init`. What do we expect it to be? Naturally, because we've negated the value of `init` once, we expect it to be `#t`. But evaluating it produces `#f`!

This problem is not an artifact of `set!`. If instead of internal mutation we had, say, printed output, the printing would have occurred twice.

To understand why, we have to examine the expanded code. It is this:

```
(let ([init #f])
  (if (begin (set! init (not init))
             init)
      (begin (set! init (not init))
             init)
      #f))
```

Aha! Because we've written the output pattern as

```
#'(if e0
      e0
      ...)
```

This looked entirely benign when we first wrote it, but it illustrates a very important principle when writing macros (or indeed any other program transformation systems): *do not copy code*! In our setting, a syntactic variable should never be repeated; if you need to repeat it in a way that might cause multiple execution of that code, make sure you have considered the consequences of this. Alternatively, if you meant to work with the *value* of the expression, bind it once and use the bound identifier's name subsequently. This is easy to demonstrate:

```
(define-syntax (my-or-4 x)
  (syntax-case x ()
    [(my-or-4)
     #'#f]
    [(my-or-4 e)
     #'e]
    [(my-or-4 e0 e1 ...)
     #'(let ([v e0])
         (if v
             v
             (my-or-4 e1 ...)))]))
```

This pattern of introducing a binding creates a new potential problem: you may end up evaluating expressions that weren't necessary. In fact, it creates a second, even more subtle one: even if it going to be evaluated, you may evaluate it in the wrong context! Therefore, you have to reason carefully about *whether* an expression will be evaluated, and if so, evaluate it once in just the right place, then store that value for subsequent use.

When we repeat our previous example, that contained the `set!`, with `my-or-4`, we see that the result is `#t`, as we would have hoped.

13.4.3 Hygiene

Hopefully now you're nervous about something else.

Do Now!

What?

Consider the macro `(let ([v #t]) (my-or-4 #f v))`. What would we expect this to compute? Naturally, `#t`: the first branch is `#f` but the second is `v`, which is bound to `#t`. But let's look at the expansion:

```
(let ([v #t])
  (let ([v #f])
    (if v
        v
        v)))
```

This expression, when run directly, evaluates to `#f`. However, `(let ([v #t]) (my-or-4 #f v))` evaluates to `#t`. In other words, the macro seems to magically produce the right value: the names of identifiers chosen in the macro seem to be independent of those introduced by the macro! This is unsurprising when it happens in a *function*; the macro expander enjoys a property called *hygiene* that gives it the same property.

One way to think about hygiene is that it effectively automatically renames all bound identifiers. That is, it's as if the program expands as follows:

```
(let ([v #t])
  (or #f v))
```

turns into

```
(let ([v1 #t])
  (or #f v1))
```

(notice the *consistent* renaming of `v` to `v1`), which turns into

```
(let ([v1 #t])
  (let ([v #f])
       v
       v1))
```

which, after renaming, becomes

```
(let ([v1 #t])
  (let ([v2 #f])
       v2
       v1))
```

when expansion terminates. Observe that each of the programs above, if run directly, will produce the correct answer.

13.5 Identifier Capture

Hygienic macros address a routine and important pain that creators of syntactic sugar confront. On rare instances, however, a developer wants to intentionally break hygiene. Returning to objects, consider this input program:

```
(define os-1
  (object/self-1
   [first (x) (msg self 'second (+ x 1))]
   [second (x) (+ x 1)]))
```

What does the macro look like? Here's an obvious candidate:

```
(define-syntax object/self-1
  (syntax-rules ()
    [(object [mtd-name (var) val] ...)
     (let ([self (lambda (msg-name)
                   (lambda (v) (error 'object "nothing here")))])
       (begin
         (set! self
               (lambda (msg)
                 (case msg
                   [(mtd-name) (lambda (var) val)]
                   ...)))
         self))]))
```

Unfortunately, this macro produces the following error:
`self: unbound identifier in module in: self`
which is referring to the `self` in the body of the method bound to `first`.

Exercise

> Work through the hygienic expansion process to understand why error is the expected outcome.

Before we jump to the richer macro, let's consider a variant of the input term that makes the binding explicit:

```
(define os-2
  (object/self-2 self
   [first (x) (msg self 'second (+ x 1))]
   [second (x) (+ x 1)]))
```

The corresponding macro is a small variation on what we had before:

```
(define-syntax object/self-2
  (syntax-rules ()
    [(object self [mtd-name (var) val] ...)
     (let ([self (lambda (msg-name)
                   (lambda (v) (error 'object "nothing here")))])
       (begin
         (set! self
               (lambda (msg)
                 (case msg
                   [(mtd-name) (lambda (var) val)]
                 ...)))
         self))]))
```

This macro expands without difficulty.

Exercise

> Work through the expansion of this version and see what's different.

This offers a critical insight: *had the identifier that goes in the binding position been written by the macro user*, there would have been no problem. Therefore, we want to be able to *pretend* that the introduced identifier was written by the user. The function `datum->syntax` converts the s-expression in its second argument; its first argument is which syntax to pretend it was a part of (in our case, the original macro use, which is bound to `x`). To introduce the result into the environment used for expansion, we use `with-syntax` to bind it in that environment:

```
(define-syntax (object/self-3 x)
  (syntax-case x ()
    [(object [mtd-name (var) val] ...)
     (with-syntax ([self (datum->syntax x 'self)])
        #'(let ([self (lambda (msg-name)
                        (lambda (v) (error 'object "nothing here")))])
            (begin
              (set! self
                    (lambda (msg-name)
                      (case msg-name
                        [(mtd-name) (lambda (var) val)]
                        ...)))
              self))))]))
```

With this, we can go back to having `self` be implicit:

```
  (define os-3
    (object/self-3
     [first (x) (msg self 'second (+ x 1))]
     [second (x) (+ x 1)]))
```

13.6 Influence on Compiler Design

The use of macros in a language's definition has an impact on all tools, especially compilers. As a working example, consider `let`. `let` has the virtue that it can be compiled efficiently, by just extending the current environment. In contrast, the expansion of `let` into function application results in a much more expensive operation: the creation of a closure and its application to the argument, achieving effectively the same result but at the cost of more time (and often space).

This would seem to be an argument against using the macro. However, a smart compiler recognizes that this pattern occurs often, and instead internally effectively converts left-left-lambda [REF] back into the equivalent of `let`. This has two advantages. First, it means the language designer can freely use macros to obtain a smaller core language, rather than having to trade that off against the execution cost.

It has a second, much subtler, advantage. Because the compiler recognizes this pattern, *other* macros can also exploit it and obtain the same optimization; they don't need to contort their output to insert `let` terms if the left-left-lambda pattern occurs naturally, as they would have to do otherwise. For instance, the left-left-lambda pattern occurs naturally when writing certain kinds of pattern-matchers, but it would take an extra step to convert this into a `let` in the expansion—which is no longer necessary.

13.7 Desugaring in Other Languages

Many modern languages define operations via desguaring, not only Racket. In Python, for instance, iterating using `for` is simply a syntactic pattern. A developer who writes `for x in o` is

- introducing a new identifier (call it `i`—but be sure to not capture any other `i` the programmer has already defined, i.e., bind `i` hygienically!),
- binding it to an iterator obtained from `o`, and