Macros and Languages in Racket

Version 2-0609.19

Ryan Culpepper <ryanc@racket-lang.org>

June 9, 2022

1 Introduction

This section introduces the elements of macro design and illustrates these design elements with simple example macro. It uses the example to introduce some of Racket's facilities for specifying, implementing, and testing macros.

This guide assumes that you have a basic working knowledge of Racket and functional programming. *The Racket Guide* is sufficient for the former, and *HtDP* is good for the latter.

1.1 How to Design Macros

This guide is an attempt to adapt the ideas of *How to Design Programs (HtDP)* to the design of macros and languages in Racket. The central idea of *HtDP* is the "design recipe"; the kernel of the design recipe consists of the following four steps:

- Specify the inputs.
- Write examples that can be turned into tests.
- Choose an implementation strategy.
- Finish the implementation and check it.

HtDP instantiates this kernel to teach the foundations of programming. Its specification language is a semiformal language of types including set-based reasoning and parametric polymorphism. Its implementation strategies include structural recursion and case analysis following data type definitions. It instantiates the implementation language to a series of simple Scheme-like functional programming languages, and it provides a testing framework.

Along the way, *HtDP* fills in the design recipe's skeleton with idioms, tricks, preferences, and limitations of Scheme-like (and ML-like) mostly-functional programming languages. For example, it demonstrates abstraction via parametric polymorphism and higher-order functions rather than OO patterns. To name some of the limitations: it uses lexical scoping; it avoids reflection (eg, no accessing structure fields by strings); it avoids `eval`; it treats closures as opaque; it (usually) avoids mutation; and so on. Once you absorb them, these parts of the programming mental model tend to be invisible, until you compare with a language that makes different choices.

This guide instantiates the design recipe kernel as follows: It introduces a specification language called *shapes*, combining features of grammars, patterns, and types. The implementation strategies are more specialized, but they are still organized around the shapes of macro inputs. The implementation language is Racket with `syntax/parse` and some other standard syntax libraries.

Along the way, it covers some of the idioms and limitations of the programming model for macros: macros (usually) respect lexical scoping; they must respect the "phase" separation between compile time and run time; they avoid `eval`; they (usually) treat expressions as opaque; and so on.

1.2 Designing Your First Macro

Suppose we wanted a feature, `assert`, that takes an expression and evaluates it, raising an error that includes the expression text if it does not evaluate to a true value. The result of the `assert` expression itself is `(void)`.

Clearly, `assert` cannot be a function, because a function cannot access the text of its arguments. It must be a macro.

We can specify the *shape* of `assert` as follows:

```
;; (assert Expr) : Expr
```

That is, the `assert` macro takes a single argument, an expression, and a use of the `assert` macro is an expression.

Here are some examples that illustrate the intended behavior of `assert`:

```
> (define ls '(1 2 3))
> (assert (> (length ls) 2))
> (assert (even? (length ls)))
assert: assertion failed: (even? (length ls))
```

In addition to considering the macro's behavior, it can be useful to consider what code could be used to implement an example use of the macro. The second example, for instance, could be implemented by the following code:

```
(unless (even? (length ls))
  (error 'assert "assertion failed: (even? (length ls))"))
```

It would be a bit complicated (although possible) for our `assert` macro to produce this exact code, because it incorporates the argument expression into a string literal. But there's no need to produce that string literal at compile time. Here is an equivalent bit of code that produces the same string at run time instead, with the help of `quote` and `error`'s built-in formatting capabilities:

```
(unless (even? (length ls))
  (error 'assert "assertion failed: ~s" (quote (even? (length ls)))))
```

Lesson: *Don't fixate on the exact code you first write down for the macro's example expansion. Often, you can change it slightly to make it easier for the macro to produce.*

Lesson: *It's often simpler to produce an expression that does a computation at run time than to do the computation at compile time.*

That's our implementation strategy for the `assert` macro: we will simply use `unless`, `quote`, and `error`. In general, the macro performs the following transformation:

```
(assert condition)
⇒
(unless condition
  (error 'assert "assertion failed: ~s" (quote condition)))
```

Before we define the macro, we must import the machinery we'll use in its implementation:

```
(require (for-syntax racket/base syntax/parse))
```

The `for-syntax` modifier indicates that we need these imports to perform *compile-time* computation — a macro is implemented by a compile-time function from syntax to syntax. We need `racket/base` for syntax templates. We need `syntax/parse` for `syntax-parser`, which is a pattern-matching utility for syntax objects.

Here is the macro definition:

```
; (assert Expr) : Expr
(define-syntax assert
  (syntax-parser
    [(_ condition:expr)
     (syntax (unless condition
               (error 'assert "assertion failed: ~s"
                      (quote condition))))]))
```

Here is an overview of the macro definition:

- The macro is defined using `define-syntax`, which takes the macro's name and a *compile-time* expression for the macro's *transformer* function. By "compile-time expression", I mean that the expression is evaluated at compile time using the *compile-time environment*, which is distinct from the normal environment. We initialized the compile-time environment earlier with `(require (for-syntax racket/base syntax/parse))`.
- The transformer takes a syntax object representing the macro use and returns a syntax object for the macro's expansion. This transformer is implemented with `syntax-parser`, which takes a sequence of clauses consisting of a *syntax pattern* and a *result expression*. This macro's transformer has only one clause.

- The pattern `(_ condition:expr)` says that after the macro name (typically represented by the wildcard pattern `_`) the macro expects one expression, representing a "condition". The identifier `condition` is a *syntax pattern variable*; it is *annotated* with the *syntax class* `expr`. If the clause's pattern matches the macro use, then its pattern variables are defined and available in *syntax templates* in the clause's result expression.

- The clause's result expression is a `syntax` expression, which contains a *syntax template*. It is similar to `quasiquote` except that pattern variables do not need explicit `unquote`s. (It also cooperates with ellipses and some other features; we'll talk about them later.) When the `syntax` expression is evaluated, it produces a syntax object with the pattern variables in the template replaced with the terms matched from the macro use. Note that even the occurrence within the `quote` term gets replaced — pattern variable substitution happens before the `quote` is interpreted, so a `quote` in the template is treated like any other identifier.

Finally, we should test the macro. I'll use `rackunit` for testing:

```
> (require rackunit)
```

Here `rackunit` is required normally, not `for-syntax`, because I intend to use it to test the behavior of `assert` expressions; I don't intend to test `assert`'s compile-time transformer function directly.

```
> (define ls '(1 2 3))
> (check-equal? (assert (> (length ls) 1))
                (void))
> (check-exn exn:fail?
             (lambda ()
               (assert (even? (length ls)))))
```

What if we want to test uses of `assert` that might result in compile-time exceptions, like syntax errors? The following does not work:

```
> (check-exn exn:fail:syntax?
             (lambda ()
               (assert (odd? (length ls)) 'an-extra-argument)))
eval:12:0: assert: unexpected term
  at: (quote an-extra-argument)
  in: (assert (odd? (length ls)) (quote an-extra-argument))
```

Racket expands and compiles expressions before it evaluates them. The syntax error is detected and raised at compile time (during expansion), but `check-exn` does not install its exception handler until run time.

One solution is to use eval for this test. This is one of the few “good” uses of eval in Racket programming. Here’s one way to do it:

```
> (define-namespace-anchor anchor)
> (check-exn exn:fail:syntax?
             (lambda ()
               (parameterize ((current-namespace (namespace-
anchor->namespace anchor)))
                 (eval #'(assert (odd? (length ls)) 'an-extra-
argument)))))
```

Another solution is to catch the compile-time exception and “save it” until run time. The `syntax/macro-testing` library has a form called `convert-syntax-error` that does that:

```
> (require syntax/macro-testing)
> (check-exn exn:fail:syntax?
             (lambda ()
               (convert-syntax-error
                (assert (odd? (length ls)) 'an-extra-argument))))
```

That completes the design of the `assert` macro. We covered specification, examples, implementation strategy, implementation, and testing.

1.3 Expansion Contexts and Expansion Order

Consider the shape of `assert`:

```
;; (assert Expr) : Expr
```

The first `Expr` is for the macro’s argument. The second `Expr`, though, says that `assert` forms a new kind of expression. But this also points to a limitation of macros: `assert` is *only* a new kind of expression.

Not every term in a program matching a macro’s pattern is expanded (that is, rewritten). Macros are expanded only in certain positions, called *expansion contexts*—essentially, contexts where expressions or definitions may appear. For example, if `assert` is the macro defined above, then the following occurrences of `assert` do *not* count as uses of the macro, and they don’t get expanded:

- `(let ((assert (> 1 2))) 'ok)` — This occurrence of `assert` is in a `let`-binding; `assert` is interpreted as a variable name to bind to the value of `(> 1 2)`. In Racket, names like `lambda`, `if`, and `assert` can be shadowed just like variables can!

- `(cond [assert (odd? 4)] [else 'nope])` — This is a syntax error. The `cond` form treats `assert` and `(odd? 4)` as separate expressions, and the use of `assert` as an expression by itself is a syntax error (the use does not match `assert`'s pattern).
- `'(assert #f)` — This `assert` occurs as part of a `quote`d constant.

Note that `let` and `cond` are also macros. So we cannot even tell whether a term involving `assert` is used as an expression until we understand the shapes of the surrounding macros. In particular, the Racket macro expander expands macros in "outermost-first" order, in contrast to nested function calls, which are evaluated "innermost-first." The outermost-first expansion order is necessary because the macro expander only knows the shapes (and thus the expansion contexts) of primitive syntactic forms; it must expand away the outer macros so that it knows what inner terms need to be expanded.

1.4 Proper Lexical Scoping

Given that `assert` just expands into uses of `unless`, `error`, and so on, perhaps we could interfere with its intended behavior by locally shadowing names it depends on — `error`, for example. But if we try it, we can see it has no effect:

```
> (let ([error void])
    (assert (even? (length ls))))
```

assert: assertion failed: (even? (length ls))

The `assert` macro is *properly lexically scoped*, or *hygienic*. Roughly, that means that references in `assert`'s syntax template are resolved in the environment where the macro was defined, not the environment where it is used. This is analogous to the behavior you would get if `assert` were a function: functions automatically close over their free variables. In the case of macros, it is syntax objects that contain information about the syntax's *lexical context*.

In other words, the following "naive" code is the wrong explanation for the expansion of this `assert` example:

```
; WRONG
(let ([error void])
  (unless (even? (length ls))
    (error 'assert "assertion failed: ~s" (quote (even? (length ls))))))
```

Instead, each term introduced by `assert` carries some lexical context information with it. Here's a better way to think of the expansion:

```
(let ([error void])
  (unlessᵐ (even? (length ls))
    (errorᵐ 'assert "assertion failed: ~s" (quoteᵐ (even? (length ls))))))
```

The lexical contexts of error and error^{m} prevents the use-site local binding of error from capturing the reference error^{m}.

This example illustrates one half of *hygienic macro expansion*. We'll talk about the other half in §3.3 "Proper Lexical Scoping, Part 2".

1.5 More Implementations of assert

Given that we have all of "ordinary" Racket plus several different macro-defining DSLs available for the implementation of assert's transformer function, there are many other ways we could implement it. This section introduces a few of them.

A (syntax *template*) expression can be written as #'*template* instead. That is, #' is a reader macro for syntax. So the assert macro can be defined as follows:

```
; (assert Expr) : Expr
(define-syntax assert
  (syntax-parser
    [(_ condition:expr)
     #'(unless condition
         (error 'assert "assertion failed: ~s"
                (quote condition)))]))
```

The syntax-parser form is basically a combination of lambda and syntax-parse. So the following definition is equivalent:

```
; (assert Expr) : Expr
(define-syntax assert
  (lambda (stx)
    (syntax-parse stx
      [(_ condition:expr)
       #'(unless condition
           (error 'assert "assertion failed: ~s"
                  (quote condition)))])))
```

The define-syntax form supports "function definition" syntax like define does, so the following is also allowed:

```
; (assert Expr) : Expr
(define-syntax (assert stx)
  (syntax-parse stx
    [(_ condition:expr)
     #'(unless condition
         (error 'assert "assertion failed: ~s"
                (quote condition)))]))
```

A macro's transformer function is, in a sense just an ordinary Racket function, except that it exists at compile time. When we imported (for-syntax racket/base) earlier, we made the Racket language available at compile time. We can define the transformer as a separate compile-time function using begin-for-syntax; the definitions it contains are added to the compile-time environment. Then we can simply use a reference to the function as the implementation of assert.

```
(begin-for-syntax
  ; assert-transformer : Syntax[(_ Expr)] -> Syntax[Expr]
  (define (assert-transformer stx)
    (syntax-parse stx
      [(_ condition:expr)
       #'(unless condition
           (error 'assert "assertion failed: ~s"
                  (quote condition)))])))
; (assert Expr) : Expr
(define-syntax assert assert-transformer)
```

Note: There are two differences between assert and assert-transformer. The name assert is bound as a *macro* in the *normal environment* (also called the *run-time environment* or the *phase-0 environment*), whereas the name assert-transformer is bound as a *variable* in the *compile-time environment* (also called the *transformer environment* or the *phase-1 environment*). Both of them are associated with a compile-time value, but assert-transformer is not a macro; if you replace assert with assert-transformer in the tests above, they will not even compile. Likewise, you cannot use assert in a compile-time expression, either as a macro or as a variable. The separation of run-time and compile-time environments is part of Racket's *phase separation*.

In addition to syntax/parse, Racket also inherits Scheme's older macro-definition DSLs: syntax-rules and syntax-case, and they are used in much existing Racket code. Here are versions of assert written using those systems:

```
(define-syntax-rule (assert condition)
  (unless condition
    (error 'assert "assertion failed: ~s" (quote condition))))

(define-syntax assert
  (syntax-rules ()
    [(_ condition)
     (unless condition
       (error 'assert "assertion failed: ~s" (quote condition)))]))

(define-syntax (assert stx)
  (syntax-case stx ()
    [(_ condition)
     #'(unless condition
```

```
        (error 'assert "assertion failed: ~s"
               (quote condition)))]))
```

For a macro as simple as assert, there isn't much difference. All of the systems share broadly similar concepts such as syntax patterns and templates. The syntax/parse system evolved out of syntax-case; syntax/parse adds a more sophisticated pattern language and a more expressive way of organizing compile-time syntax validation and computation.

All of these pattern-matching DSLs are simply aids to writing macros; they aren't necessary. It's possible to write the macro by directly using the syntax object API. Here's one version:

```
(define-syntax assert
  (lambda (stx)
    (define parts (syntax->list stx)) ; (U (Listof Syntax) #f)
    (unless (and (list? parts) (= (length parts) 2))
      (raise-syntax-error #f "bad syntax" stx))
    (define condition-stx (cadr parts)) ; Syntax[Expr]
    (define code
      (list (quote-syntax unless) condition-stx
            (list (quote-syntax error) (quote-syntax 'assert)
                  (quote-syntax "assertion failed: ~s")
                  (list (quote-syntax quote) condition-stx))))
    (datum->syntax (quote-syntax here) code)))
```

Briefly, syntax->list unwraps a syntax object one level and normalizes it to a list, if possible (the terms (a b c) and (a . (b c)), while both "syntax lists", have different syntax object representations). It is built on top of the primitive operation syntax-e. The quote-syntax form is the primitive that creates a syntax object constant for a term that captures the lexical context of the term itself. The lexical context can be transferred to a tree using datum->syntax; it wraps pairs, atoms, etc, but it leaves existing syntax objects unchanged.

Here is a variant of the previous definition that uses quasisyntax (reader abbreviation #\`) and unsyntax (reader abbreviation #,) to construct the macro's result:

```
(define-syntax assert
  (lambda (stx)
    (define parts (syntax->list stx)) ; (U (Listof Syntax) #f)
    (unless (and (list? parts) (= (length parts) 2))
      (raise-syntax-error #f "bad syntax" stx))
    (define condition-stx (cadr parts)) ; Syntax[Expr]
    #`(unless #,condition-stx
        (error 'assert "assertion failed: ~s"
               (quote #,condition-stx)))))
```

It is not a goal of this guide to introduce you to every bit of machinery that can be used to implement macros. In general, this guide will stick to the `syntax/parse` system for macro definitions, and it uses the `#'` abbreviation for `syntax` expressions. It will sometimes be necessary to use the lower-level APIs (such as `syntax->list`, `` #` `` and `#,`) to perform auxiliary computations.

On the other hand, it is a goal of this guide to discuss and compare different implementation strategies. So the following sections do often present multiple implementations of the same macro according to different strategies.

2 Terms and Shapes

This section introduces terminology for talking about the pieces of Racket programs and their interpretation. In particular, it introduces the idea of *shapes*, which we will use as the specification language that drives macro design and organizes implementation strategies.

2.1 Terms

Consider the following Racket code:

```
(define (map f xs)
  (cond [(pair? xs)
         (cons (f (car xs)) (map f (cdr xs)))]
        [(null? xs) '()]))
```

The code is a tree of terms. A *term* is, roughly, an atom or a parenthesized group of terms. So all of the following are terms:

- `define`, `map`, `xs`, `pair?` — More specifically, these are *identifier terms*.
- `(pair? xs)`, `(f (car xs))`, `[(null? xs) '()]` — More specifically, these are *list terms*.
- `'()` — This is also a list term, because it is read as `(quote ())`.

The following is not a term:

- `map f` — That's two terms.

The following are also terms that occur in the program above, even though it might not be immediately apparent:

- `(f xs)` — Because `(map f xs)` is the same as `(map . (f xs))`, which is also the same as `(map . (f . (xs . ())))`.
- `quote` — Because it's a subterm of `'()`, which is the same as `(quote ())`.

Here are some other terms that don't appear in the program above:

- `#t`, `5`, `#e1e3`, `"racket-lang.org"`, `#(1 2 3)`, `#s(point 3 4)`, `#:unless`, `#rx"[01]+"` — A *boolean term*, two *number terms*, a *string term*, and so on.

Racket represents terms using *syntax objects*, a kind of value.

It will be helpful to keep the two levels separate (term vs value representation), but that's hard, because we don't have enough distinct terms (err, I mean words) to name everything. In some cases, the context should either make the usage clear or make the distinction moot. In some cases, I'll disambiguate by saying, for example, *identifier term* vs *identifier value*.

2.2 Interpretations of Terms

What is an expression?

The concept of "expression" doesn't simply refer to some subset of terms. *Any* term can be an expression, given the right context. And a term might be an expression when used in one place but not when used in another. Is the identifier `f` an expression? In the example code above, the first occurrence of `f` is not an expression, but the second and third occurrences are expressions. It depends on context — that is, where the term appears in the code. The term `f` isn't an expression when it occurs in the function definition's formal parameter list, but it is an expression when it occurs in operator position of an application. What is an "application"? Well, it's a kind of expression — and so we have to keep looking outward to figure out what's going on.

Here's the reasoning for the second and third occurrences of `f` being expressions: The example is a use of `define`, and the rule for `define` is that the body is an expression (that's an oversimplification, actually). The body is a `cond` expression, and a `cond` expression's arguments are "clauses", which are not expressions themselves, but consist of two expressions grouped together (again, oversimplified). The second expression of the first `cond` clause is a function call to `cons`, so its first argument is an expression. And that expression is a function call (because `f` is bound as a variable), so that `f` is an expression. And that's how we know, starting from the top.

Of course, if we wrap `quote` around the whole thing, then all of that reasoning is invalidated, because the argument of a `quote` expression is not interpreted as a definition or expression.

So "expression" doesn't refer to a subset of terms (decidable or not). But that doesn't mean that it isn't an important concept. Rather, "expression" describes an *interpretation* or *intended usage* of a term. Here are names for the main interpretations that are handled by Racket's macro expander:

- *expression* or *expression term* — Used in an expression position, like the test of an `if` or an argument to a function.
- *body term* — Used as one element of a `lambda` body, `let` body, etc. A "body" is also called an "internal definition context".
- *module-level term* — Used as one element of a `module` body or submodule body.

- *top-level term* — Used at the top level, for example at the REPL or in a call to `eval`.

The word *form* is used to identify a variant of expression, module-level term, etc. The concept of "variant" usually coincides with the leading identifier of the term. For example: `if` is an expression form; `provide` is a module-level form but not a top-level form, but `require` is allowed both as a module-level form and as a top-level form.

The word *form* can also refer to the entire term, as in "`(require racket/list)` is a module-level form".

The word *definition* refers to a subset of the body forms, roughly. In fact, we could say that a body term is either an expression or a definition.

2.3 Shapes

When we design a macro, the intended interpretation of an argument can be as important or more important than the set of terms allowed for that argument. To usefully describe macros and the ways they treat their arguments, we need to talk about both of these aspects. We'll do that with a semi-formal description language of *shapes*.

A shape has two aspects:

- the *set of terms* belonging to the shape, and
- the *interpretation* or intended usage of the terms of that shape

Different basic shapes place different degrees of emphasis on these two aspects.

A shape is not the same thing as a syntax pattern, although there is generally a correspondence between shapes and patterns. In particular, we'll use implement basic shapes using *syntax classes*. A syntax class check terms for membership in the shape's set of terms and it can compute attributes related to the interpretation of the shape. But a syntax class cannot always check every aspect of a shape's interpretation; for example, a syntax class cannot verify that we use a term in an expression position in the code that we generate. That obligation stays with the macro writer.

The following sections introduce different shapes and show how they affect the design and implementation of macros that use them.

3 Basic Shapes

This section introduces the most important basic shapes for macro design.

3.1 The Expr (Expression) Shape

The Expr shape represents the intention to interpret the term as a Racket expression by putting it in an expression context. In general, a macro cannot check a term and decide whether it is a valid expression; only the Racket macro expander can do that. As a pragmatic approximation, the Expr shape and its associated expr syntax class exclude only keyword terms, like #:when, so that macros can detect and report misuses of keyword arguments.

As an example, let's implement my-when, a simple version of Racket's when form. It takes two expressions; the first is the condition, and the second is the result to be evaluated only if the condition is true. Here is the shape:

```
;; (my-when Expr Expr) : Expr
```

Here are some examples:

```
(my-when (odd? 5) (printf "odd!\n"))    ; expect print
(my-when (even? 5) (printf "even!\n"))  ; expect no print
```

Here's the implementation:

```
(define-syntax my-when
  (syntax-parser
    [(_ condition:expr result:expr)
     #'(if condition result (void))]))
```

We use the expr syntax class to annotate pattern variables that have the Expr shape. Note that the names of the pattern variables do not include the :expr annotation, so in the syntax template we simply write condition and result.

To test the macro, we rephrase the previous examples as tests:

```
> (check-equal? (with-output-to-string
                  (lambda () (my-when (odd? 5) (printf "odd!\n"))))
                "odd!\n")
> (check-equal? (with-output-to-string
                  (lambda () (my-when (even? 5) (printf "even!\n"))))
                "")
```

```
> (check-exn exn:fail:syntax?
             (lambda ()
               (convert-syntax-error
                (my-when #:truth "verity"))))
```

Exercise 1: Each of the following uses of `my-when` violates its declared shape:

- `(my-when #:true "verity")`
- `(my-when 'ok (define ns '(1 2 3)) (length ns))`
- `(my-when (odd? 1) (begin (define one 1) (+ one one)))`
- `(my-when #f (+ #:one #:two))`

Why? Which examples are rejected by the `my-when` macro itself, and what happens to the other examples? What difference does it make if you remove the `expr` syntax class annotations from the macro definition?

Exercise 2: Design a macro `my-unless` like `my-when`, except that it negates the condition.

Exercise 3: Design a macro `catch-output` that takes a single expression argument. The expression is evaluated, but its result is ignored; instead, the result of the macro is a string containing all of the output written by the expression. For example:

```
(catch-output (for ([i 10]) (printf "~s" i)))
; expect "0123456789"
```

3.2 The Body Shape

The `Body` shape is like `Expr` except that it indicates that the term will be used in a body context, so definitions are allowed in addition to expressions.

There is no distinct syntax class for `Body`; just use `expr`.

In practice, the `Body` shape is usually used with ellipses; see §4 "Compound Shapes". But we can make a version of `my-when` that takes a single `Body` term, even though it isn't idiomatic Racket syntax. Here is the shape:

```
;; (my-when Expr Body) : Expr
```

Here is an example allowed by the new shape but not by the previous shape:

```
(define n 37)
(my-when (odd? n)
  (begin (define q (quotient n 2)) (printf "q = ~s\n" q)))
```

Given the new shape, the previous implementation would be wrong, since it does not place its second argument in a body context. Here is an updated implementation:

```
(define-syntax my-when
  (syntax-parser
    [(_ condition:expr result-body:expr)
     #'(if condition (block result-body) (void))]))
```

That is, use `(block ␣)` to wrap a `Body` so it can be used in a strict `Expr` position. It is also common to use a `(let () ␣)` wrapper, but that does not work for all `Body` terms; it requires that the `Body` term ends with an expression. The `block` form is more flexible.

Racket's `#%expression` form is useful in the opposite situation. It has the following shape:

```
;; (#%expression Expr) : Body
```

That is, use `(#%expression ␣)` to turn a `Body` position into a strict `Expr` position.

Exercise 4: Check your solution to Exercise 3; does the macro also accept `Body` terms like the one above? That is, does the following work?

```
(catch-output
 (begin (define q (quotient n 2))
        (printf "q = ~s\n" q)))
```

If so, "fix it" (that is, make it more restrictive) using `#%expression`.

3.3 Proper Lexical Scoping, Part 2

Here is one solution to Exercise 3 using `with-output-to-string`:

```
; (catch-output Expr) : Expr
(define-syntax catch-output
  (syntax-parser
    [(_ e:expr)
     #'(with-output-to-string (lambda () (#%expression e)))]))
```

Racket already provides `with-output-to-string` from the `racket/port` library, but if it did not, we could define it as follows:

```
; with-output-to-string : (-> Any) -> String
(define (with-output-to-string proc)
  (let ([out (open-output-string)])
    (parameterize ((current-output-port out))
      (proc))
    (get-output-string out)))
```

Here is another implementation of catch-output, which essentially inlines the definition of with-output-to-string into the macro template:

```
; (catch-output Expr) : Expr
(define-syntax catch-output
  (syntax-parser
    [(_ e:expr)
     #'(let ([out (open-output-string)])
         (parameterize ((current-output-port out))
           (#%expression e))
         (get-output-string out))]))
```

In §1.4 "Proper Lexical Scoping" we saw that we cannot interfere with a macro's "free variables" by shadowing them at the macro use site. For example, the following attempt to capture the macro's reference to get-output-string fails:

```
> (let ([get-output-string (lambda (p) "pwned!")])
    (catch-output (printf "doing just fine, actually")))
"doing just fine, actually"
```

But what about the other direction? The macro introduces a binding of a variable named out; could this binding capture references to out in the expression given to the macro? Here is an example:

```
> (let ([out "Aisle 24"])
    (catch-output (printf "The exit is located at ~a." out)))
"The exit is located at Aisle 24."
```

The result shows that the macro's out binding does not interfere with the use-site's out variable. We say that the catch-output macro is "hygienic".

A macro is *hygienic* if it follows these two lexical scoping principles:

1. A *use-site binding* does not capture a *definition-site reference*.
2. A *definition-site binding* does not capture a *use-site reference*.

Racket macros are hygienic by default. In FIXME-REF we will discuss a few situations when it is useful to break hygiene.

3.4 The Identifier Shape

The Id shape contains all identifier terms.

The `Id` shape usually implies that the identifier will be used as the name for a variable, macro, or other sort of binding. In that case, we say the identifer is used as a *binder*.

Use the `id` syntax class for pattern variables whose shape is `Id`.

Let's write a macro `my-and-let` that acts like `and` with two expressions but binds the result of the first expression to the given identifier before evaluating the second expression. Here is the shape:

```
;; (my-and-let Id Expr Expr) : Expr
```

Here are some examples:

```
(define ls '((a 1) (b 2) (c 3)))
(my-and-let entry (assoc 'b ls) (cadr entry)) ; expect 2
(my-and-let entry (assoc 'z ls) (cadr entry)) ; expect #f
```

Here is an implementation:

```
(define-syntax my-and-let
  (syntax-parser
    [(_ x:id e1:expr e2:expr)
     #'(let ([x e1]) (if x e2 #f))]))
```

The main point of `my-and-let`, though, is that if the second expression is evaluated, it is evaluated in an environment where the identifier is bound to the value of the first expression. Let's put that information in the shape of `my-and-let`. It requires two changes:

- Label the identifier so we can refer to it later. So instead of `Id`, we write `x:Id`. The label does not have to be the same as the name of the pattern variable, but it makes sense to use the same name here.
- Add an *environment annotation* to the second `Expr` indicating that it is in the scope of a variable whose name is whatever actual identifier `x` refers to: `Expr{x}`.

Here is the updated shape for `my-and-let`:

```
;; (my-and-let x:Id Expr Expr{x}) : Expr
```

We can check the implementation: `e1` does not occur in the scope of `x`, and `e2` does occur in the scope of `x`.

Here is another implementation:

```
; (my-and-let x:Id Expr Expr{x}) : Expr
(define-syntax my-and-let
  (syntax-parser
    [(_ x:id e1:expr e2:expr)
     #'(let ()
         (define x e1)     ; BAD
         (if x e2 #f))]))
```

This implementation is *wrong*, because `e1` occurs in the scope of `x`, but it should not.

Here is another version:

```
; (my-and-let x:Id Expr Expr{x}) : Expr
(define-syntax my-and-let
  (syntax-parser
    [(_ x:id e1:expr e2:expr)
     #'(let ()
         (define tmp e1)
         (if tmp (let ([x tmp]) e2) #f))]))
```

This implementation is good (although more complicated than unnecessary), because `e1` no longer occurs in the scope of `x`. But what about `tmp`? Because of hygiene, the definition of `tmp` introduced by the macro is not visible to `e1`. (To be clear, it would be *wrong* to write `Expr{tmp}` for the shape of the first expression.)

Exercise 5: Generalize `my-and-let` to `my-if-let`, which takes an extra expression argument which is the macro's result if the condition is false. The macro should have the following shape:

```
;; (my-if-let x:Id Expr Expr{x} Expr) : Expr
```

Double-check your solution to make sure it follows the scoping specified by the shape.

3.5 Expressions, Types, and Contracts

Let's design the macro `my-match-pair`, which takes an expression to destructure, two identifiers to bind as variables, and a result expression. Here are some examples:

```
(my-match-pair (list 1 2 3) n ns (< n (length ns)))
; expect #t
(my-match-pair (list 'p "hello world") tag content
               (format "<~a>~a</~a>" tag (string-join content "
") tag))
; expect "<p>hello world</p>"
```

Here is one shape we could write for `my-match-pair`:

```
;; (my-match-pair Expr x:Id xs:Id Expr{x,xs}) : Expr
```

Here's an implementation:

```
(define-syntax my-match-pair
  (syntax-parser
    [(_ pair:expr x:id xs:id result:expr)
     #'(let ([pair-v pair])
         (let ([x (car pair-v)]
               [xs (cdr pair-v)])
           result))]))
```

Note that we introduce a *temporary variable* (or *auxiliary variable*) named `pair-v` to avoid evaluating the `pair` expression twice.

We could add more information to the shape. The macro expects the first argument to be a pair, and whatever types of values the pair contains become the types of the identifiers:

```
;; (my-match-pair Expr[(cons T1 T2)] x:Id xs:Id Expr{x:T1,xs:T2}) : Expr
```

I've written `Expr[(cons T1 T2)]` for the shape of expressions of type `(cons T1 T2)`, where the type `(cons T1 T2)` is the type of all pairs (values made with the `cons` constructor) whose first component has type `T1` and whose second component has type `T2`. The second expression's environment annotation includes the types of the variables. This macro shape is polymorphic; there is an implicit `forall (T1, T2)` at the beginning of the declaration.

The result of the macro is the result of the second expression, so the type of the macro is the same as the type of the second expression. We could add that to the shape too:

```
;; (my-match-pair Expr[(cons T1 T2)] x:Id xs:Id Expr{x:T1,xs:T2}[R]) : Expr[R]
```

Now the second `Expr` has both a environment annotation and a type annotation.

When I say "type" here, I'm not talking about Typed Racket or some other typed language implemented in Racket, nor do I mean that there's a super-secret type checker hidden somewhere in Racket next to a flight simulator. By "type" I mean a semi-formal, unchecked description of expressions and macros that manipulate them. In this case, the shape declaration for `my-match-pair` warns the user that the first argument must produce a pair. If it doesn't, the user has failed their obligations, and the macro may do bad things.

Of course, given human limitations, we would prefer the macro not to do bad things. Ideally, the macro definition and macro uses could be statically checked for compliance with shape declarations, but Racket does not not implement such a checker for macros. (It's complicated.) At least, though, the macro enforce approximations of the types of expression arguments using *contracts*.

Use the `expr/c` syntax class for a pattern variable whose shape is `Expr[Type]` when `Type` has a useful contract approximation. In this example, the type `(cons T1 T2)` has a useful contract approximation `pair?`, but there is no useful contract for the type `R`. The `expr/c` syntax class takes an argument, so you cannot use the `:` notation; you must use `~var` or `#:declare` instead. The argument is a syntax object representing the contract to apply to the expression. (It is `#'pair?` instead of `pair?` because the contract check is performed at run time.) In the syntax template, use the `c` ("contracted") *attribute* of the pattern variable to get the expression with a contract-checking wrapper. Here's the contract-checked version of the macro:

```
; (my-match-pair Expr[(cons T1 T2)] x:Id xs:Id
Expr{x:T1,xs:T2}[R]) : Expr[R]
(define-syntax my-match-pair
  (syntax-parser
    [(_ (~var pair (expr/c #'pair?)) x:id xs:id result:expr)
     #'(let ([pair-v pair.c])   ; Important: pair.c, not pair
         (let ([x (car pair-v)]
               [xs (cdr pair-v)])
           result))]))
```

Here's the implementation using `#:declare` instead of `~var`:

```
(define-syntax my-match-pair
  (syntax-parser
    [(_ pair x:id xs:id result:expr)
     #:declare pair (expr/c #'pair?)
     #'(let ([pair-v pair.c])   ; Important: pair.c, not pair
         (let ([x (car pair-v)]
               [xs (cdr pair-v)])
           result))]))
```

Now calling `my-match-pair` raises a contract violation if the first expression does not produce a pair. For example:

```
> (my-match-pair 'not-a-pair n ns (void))
my-match-pair: contract violation
  expected: pair?
  given: 'not-a-pair
  in: pair?
      macro argument contract
  contract from: top-level
  blaming: top-level
```

(assuming the contract is correct)
at: eval:32:0

Exercise 6: Modify the `my-when` macro to check that the condition expression produces a boolean value. (Note: this is not idiomatic for Racket conditional macros).

3.6 Uses of Expressions

In general, what can a macro do with an expression (`Expr`)?

- It can use the value (or values) that the expression evaluates to. For example, the behavior of the `my-when` macro depends on the value that its first expression produces.
- It can determine whether the expression is evaluated or when the expression is evaluated. The `my-when` example determines whether to evaluate its second expression. The standard `delay` macro is a classic example of controlling when an expression is evaluated.
- It can change what dynamic context the expression is evaluated within. For example, a macro could use `parameterize` to evaluate the expression in a context with different values for some parameters.
- It can change the static context the expression is evaluated within. Mainly, this means putting the expression in the scope of additional bindings, as we did in `my-and-let` and `my-match-pair`.

There are some restrictions on what macros can do and should do with expressions:

- **A macro cannot get the value of the expression at compile time.** The expression represents computation that will occur later, at run time, perhaps on different machines, perhaps many times with different values in the run-time environment. A macro can only interact with an expression's value by producing code to process the value at run time.
- **A macro must not look at the contents of the expression itself.** Expressions are macro-extensible, so there is no grammar to guide case analysis. Interpreting expressions is the macro expander's business, so don't try it yourself. The macro expander is complicated, and if you attempt to duplicate its work "just a little", you are likely to make unjustified assumptions and get it wrong. For example, an expression consisting of a self-quoting datum is not necessarily a constant, or even free of side effects; it might have a nonstandard `#%datum` binding, which could give it any behavior at all. Likewise, a plain identifier is not necessarily a variable reference; it might be an identifier macro, or it might have a nonstandard `#%top` binding.

  In later sections (§??? "[missing]"), we'll talk about how to cooperate with the macro expander to do case analysis of expressions and other forms.

- In general, a macro should not duplicate an argument expression. That is, the expression should occur exactly once in the macro's expansion. Duplicating expressions leads to expanding the same code multiple times, which can lead to slow compilation and bloated compiled code. The increases to both time and code size are potentially exponential, if duplicated expressions themselves contain macros that duplicate expressions and so on.

  If you need to refer to an expression's value multiple times, bind it to a temporary variable. If you need to evaluate the same expression multiple times, then bind a temporary variable to a thunk containing the expression and then apply the thunk multiple times.

  One exception to this rule is if the macro knows that the expression is "simple", like a variable reference or quoted constant, because the macro is private and all of its use sites can be inspected. We'll discuss this case in §5.3 "Helper Macros and Simple Expressions".

- In general, a macro should evaluate expressions in the same order that they appear (that is, "left to right"), unless it has a reason to do otherwise.

  In Racket information generally flows from left to right, and the interpretation of later terms can depend on earlier terms. For example, `my-when` uses the value of its first (that is, left-most) expression argument to decide whether to evaluate its second (that is, right-most) expression. It would be non-idiomatic syntax design to put the condition expression second and the result expression first.

  Similarly, the scope of an identifier is generally somewhere to the right of the identifier itself. For example, in `my-match-pair`, the identifiers are in scope in the following expression. If we swapped `my-match-pair`'s expressions, so it had the shape `(my-match-pair Expr{x,xs} x:Id xs:Id Expr)`, that would not be idiomatic.

The same principles apply to `Body` terms as well.

4 Compound Shapes

This section introduces compound shapes, including list shapes and ellipsis shapes. It discusses several implementation strategies for macros consuming ellipsis shapes.

4.1 List Shapes

The main kind of *compound shape* is the *list shape*, describing list terms of fixed or varying length. Actually, we have already been using list shapes to describe a macro's arguments: a macro transformer function in fact receives exactly one argument, corresponding to the whole macro-use term. Typically, that is a list term with the macro identifier first and the arguments making up the rest of the list.

We can add additional levels of grouping to the arguments. For example, here's a variant of `my-and-let` that groups the identifier with the expression that provides its value:

```
; (my-and-let2 [x:Id Expr] Expr{x}) : Expr
(define-syntax my-and-let2
  (syntax-parser
    [(_ [x:id e1:expr] e2:expr)
     #'(let ([x e1])
          (if x e2 #f))]))
```

By itself, though, this change isn't very interesting. The real utility of list shapes (and patterns, and templates) is in their interaction with enumeration shapes and ellipses. We'll discuss ellipses now as a special case and discuss enumeration shapes later.

4.2 Ellipses with Simple Shapes

Ellipses mean zero or more repetitions of the preceding shape, pattern, or template. They are like the star (*) operator in regular expressions. For example, here is the shape of Racket's `and` macro:

```
;; (and Expr ...) : Expr
```

How can we implement our own macro with this shape? There are three basic implementation strategies:

- recursive macro
- recursive run-time helper function
- recursive compile-time helper function

4.2.1 Recursive Macros

The first strategy is to write a macro that does case analysis and explicit recursion — that is, the macro expands into another use of itself. Here is a recursive implementation of `my-and`:

```
; (my-and Expr ...) : Expr
(define-syntax my-and
  (syntax-parser
    [(_)
     #'#t]
    [(_ e1:expr e:expr ...)
     #'(if e1 (my-and e ...) #f)]))
```

(This isn't quite like Racket's `and`, which returns the value of the last expression if all previous expressions were true, and it evaluates the last expression in tail position. But it's close enough to illustrate ellipses and recursive macros.)

This macro divides one shape into two patterns: zero expressions or at least one expression. If we use `my-and` as follows:

```
(my-and (odd? 1) (even? 2) (odd? 3))
```

then the first pattern fails to match, but the second pattern matches with `e1` = `(odd? 1)` and `e ...` = `(even? 2) (odd? 3)`. Note that `e` doesn't match a single term; it matches a sequence of terms, and when we use `e` in the template, we must follow it with ellipses. One expansion step rewrites this program to the following:

```
⇒ (if (odd? 1) (my-and (even? 2) (odd? 3)) #f)
```

Where once there were three, now there are only two expressions in the remaining call to `my-and`. Subsequent steps rewrite that to one, and then none, and then `my-and`'s base case matches and it disappears entirely:

```
⇒ (if (odd? 1) (if (even? 2) (my-and (odd? 3)) #f) #f)
⇒ (if (odd? 1) (if (even? 2) (if (odd? 3) (my-and) #f) #f) #f)
⇒ (if (odd? 1) (if (even? 2) (if (odd? 3) #t #f) #f) #f)
```

4.2.2 Recursive Run-time Helper Function

Another implementation strategy is to expand into another variable-arity form or function. For example, here is another definition of `my-and` that relies on Racket's `andmap` function, "thunking" to delay evaluation, and the variable-arity `list` function:

```
; (my-and Expr ...) : Expr
(define-syntax my-and
  (syntax-parser
    [(_ e:expr ...)
     #'(andmap (lambda (thunk) (thunk))
               (list (lambda () e) ...))]))
```

Note the use of `(lambda () e) ...` in the template. The pattern variable `e` occurred in front of ellipses in the pattern, so it must be used in front of ellipses in the template. But the term before the ellipses in the template isn't just `e`, it is `(lambda () e)`. This `(lambda () ␣)` wrapper gets copied for every instance of `e`:

```
(my-and 1 2 3)
⇒
(andmap (lambda (thunk) (thunk))
        (list (lambda () 1) (lambda () 2) (lambda () 3)))
```

That is, ellipses in a syntax template act like an implicit `map` over the pattern variables.

For a frequently-used, simple macro like `and`, this might not be a good implementation because of run-time overhead, but for other macros this kind of implementation might be reasonable.

Lesson: *Many macros can be decomposed into two parts: a compile-time part that adds `lambda` wrappers to handle scoping and delayed evaluation, and a run-time part that implements the computation and behavior of the macro.*

4.2.3 Recursive Compile-time Helper Function

The final strategy is to use a compile-time helper function, which handles the recursion either directly or indirectly. Here is another implementation of `my-and`, where the macro itself is not recursive but the transformer uses a recursive compile-time helper function:

```
(begin-for-syntax
  ; my-and-helper : (Listof Syntax[Expr]) -> Syntax[Expr]
  (define (my-and-helper exprs)
    (if (pair? exprs)
        #`(if #,(car exprs)
              #,(my-and-helper (cdr exprs))
              #f)
        #'#t)))
; (my-and Expr ...) : Expr
(define-syntax my-and
  (syntax-parser
```

```
    [(_ e:expr ...)
     (my-and-helper (syntax->list #'(e ...)))]))
```

The compile-time `my-and-helper` function takes a list of syntax objects representing expressions and combines them into a single syntax object representing an expression. Whenever possible, annotate the `Syntax` type with the shape of the term that the syntax object represents: for example, `Syntax[Expr]`, `Syntax[(Expr ...)]`, etc. The function uses `quasisyntax` (which has the reader abbreviation ``#` ``) to create syntax from a template that allows `unsyntax` escapes (`#,`) to compute sub-terms. The macro calls the helper with a list of syntax objects. First it uses ellipses to form the syntax list containing all of the argument expressions: `#'(e ...)`. This value has the type `Syntax[(Expr ...)]`. Then it calls `syntax->list`, which unwraps the syntax list into a list of syntax — in this case, specifically, a `(Listof Syntax[Expr])`.

Because it is defined in the transformer environment (or "at phase 1"), you cannot directly call `my-and-helper` at the REPL to explore its behavior. But you can call it using `phase1-eval` special form. Keep in mind that the whole argument to `phase1-eval` is a compile-time expression, so you cannot refer to any run-time variables. Also, `phase1-eval` must be told how to convert the phase-1 (compile-time) answer into an expression to produce a phase-0 (run-time) value. When the result type is syntax, use `quote-syntax` if you want to preserve the syntax-nature of the result; otherwise, use `quote`. For example:

```
> (phase1-eval (my-and-helper (list #'(odd? 1) #'(even? 2)))
               #:quote quote-syntax)
#<syntax:eval:11:0 (if (odd? 1) (if (even? 2) #t #f) #f)>
> (phase1-eval (my-and-helper (list #'(odd? 1) #'(even? 2)))
               #:quote quote)
'(if (odd? 1) (if (even? 2) #t #f) #f)
```

The compile-time helper function could be written more concisely using `foldr`:

```
(begin-for-syntax
  ; my-and-helper : (Listof Syntax[Expr]) -> Syntax[Expr]
  (define (my-and-helper exprs)
    (foldr (lambda (expr rec-code)
             #`(if #,expr #,rec-code #f))
           #'#t
           exprs)))
```

Or we could just use the body of the compile-time helper function (now that we have eliminated the recursion) directly in the macro:

```
; (my-and Expr ...) : Expr
(define-syntax my-and-helper
  (syntax-parser
```

```
    [(_ e:expr ...)
     (foldr (lambda (expr rec-code)
              #`(if #,expr #,rec-code #f))
            #'#t
            (syntax->list #'(e ...)))]))
```

We can still use phase1-eval to explore more complicated compile-time expressions:

```
> (phase1-eval (foldr (lambda (expr rec-code)
                        #`(if #,expr #,rec-code #f))
                      #'#t
                      (syntax->list #'((odd? 1) (even? 2)))))
'(if (odd? 1) (if (even? 2) #t #f) #f)
```

4.3 Ellipses with Compound Shapes

Ellipses can also be used with compound shapes. For example, here is the shape of a simplified version of cond (it doesn't support => and else clauses):

```
;; (my-cond [Expr Expr] ...) : Expr
```

Here's a recursive implementation:

```
; (my-cond [Expr Expr] ...) : Expr
(define-syntax my-cond
  (syntax-parser
    [(_)
     #'(void)]
    [(_ [condition1:expr result1:expr] more ...)
     #'(if condition1
           result1
           (my-cond more ...))]))
```

Here is an implementation using a recursive run-time helper function:

```
; (my-cond [Expr Expr] ...) : Expr
(define-syntax my-cond
  (syntax-parser
    [(_ [condition:expr result:expr] ...)
     #'(my-cond-helper (list (lambda () condition) ...)
                       (list (lambda () result) ...))]))
; my-cond-helper : (Listof (-> Any)) (Listof (-> X)) -> X
; PRE: condition-thunks and result-thunks have the same length
```

```
(define (my-cond-helper condition-thunks result-thunks)
  (if (pair? condition-thunks)
      (if ((car condition-thunks))
          ((car result-thunks))
          (my-cond-helper (cdr condition-thunks)
                          (cdr result-thunks)))
      (void)))
```

Here is an implementation using a recursive helper *macro* — Racket's variadic `or` macro. It also relies on fact that `and` and `or` treat any value other than `#f` as true, and return that specific true value as their result when appropriate.

```
; (my-cond [Expr Expr] ...) : Expr
(define-syntax my-cond
  (syntax-parser
    [(_ [condition:expr result:expr] ...)
     #'((or (and condition (lambda () result))
            ...
            void))]))
```

Exercise 7: Implement `my-cond` using a compile-time helper function that takes a list of condition expressions and a list of result expressions:

```
;; my-cond-helper : (Listof Syntax[Expr]) (Listof Syntax[Expr]) -
> Syntax[Expr]
;; PRE: the two lists of expressions have the same length
```

Hint: Racket's `foldr` function is variadic.

Exercise 8: Generalize `my-and-let` so that it takes a list of identifier-and-expression clauses. That is, it should have the following shape:

```
;; (my-and-let ([Id Expr] ...) Expr) : Expr
```

The scope of each identifier includes all subsequent clauses and the final expression. Implement the macro either as a recursive macro or by using a compile-time helper function.

Exercise 9: Design a macro `my-evcase1` with the following shape:

```
;; (my-evcase1 Expr [Expr Expr] ...) : Expr
```

The macro evaluates its first argument to get the value to match. Then it tries each clause until one is selected. A clause is selected if its first expression produces a value equal to the value to match; that clause's second expression is the result of the macro. If no clause matches, the result is `(void)`.

```
(my-evcase1 (begin (printf "got a coin!\n") (* 5 5))
  [5 "nickel"]
  [10 "dime"]
  [25 "quarter"]
  [(/ 0) "infinite money!"])
; expect print once, result = "quarter"
```

5 Shape Definitions

This section shows how to define new shapes and their corresponding syntax classes.

This section also introduces a shape for simple expressions.

5.1 Defining New Shapes

Consider the shape we've given to `my-cond`:

```
;; (my-cond [Expr Expr] ...) : Expr
```

This tells us the structure of `my-cond`'s arguments, but it gives us no hook upon which to hang a description of the arguments' interpretation. Let's give it a name:

```
;; CondClause ::= [Expr Expr]   -- represent condition, result
```

Now when we describe the behavior of `my-cond`, we can separate out the structure and interpretation of `CondClause`s from the discussion of `my-cond` itself.

- The `my-cond` form takes a sequence of `CondClauses`, and that it tries each `CondClause` in order until one is selected, and the result of the `my-cond` expression is the result of the selected `CondClause`, or `(void)` if none was selected.
- A `CondClause` consists of two expressions. The first represents a condition; if it evaluates to a true value, then the clause is selected. The second expression determines the clause's result.

I typically write something like the terse comment above in the source code and include the longer, more precise version in the documentation.

We should also define a syntax class, `cond-clause`, corresponding to the new shape:

```
(begin-for-syntax
  (define-syntax-class cond-clause
    #:attributes (condition result) ; Expr, Expr
    (pattern [condition:expr result:expr])))
```

The syntax class has a single `pattern` form specifying the structure of the terms it accepts. The pattern variables are exported from the syntax class as *syntax-valued attributes*. I've written a comment after the `#:attributes` declaration with the shape of each attribute; they both contain `Expr` terms.

My convention is to use capitalized names such as `Expr`, `Id`, and `CondClause` for shapes and lower-case names such as `expr`, `id`, and `cond-clause` for syntax classes. Distinguishing them serves as a reminder that syntax classes represent some but not all of the meaning of shapes, just like Racket's contracts capture some but not all of the meaning of types. The syntax class checks that terms have the right structure, and its attribute names hint at their intended interpretation, but the syntax class cannot enforce that interpretation.

We update the macro's shape to refer to the new shape name, and we update the implementation's pattern to use a pattern variable annotated with the new syntax class (`c:cond-clause`). In the template, we refer to the pattern variable's *attributes* defined by the syntax class (`c.condition` and `c.result`).

```
; (my-cond CondClause ...) : Expr
(define-syntax my-cond
  (syntax-parser
    [(_)
     #'(void)]
    [(_ c:cond-clause more ...)
     #'(if c.condition
           c.result
           (my-cond more ...))]))
```

In addition to improved organization, another benefit of defining `cond-clause` as a syntax class is that `my-cond` now automatically uses `cond-clause` to help explain syntax errors. For example:

```
> (my-cond 5)
eval:9:0: my-cond: expected cond-clause
  at: 5
  in: (my-cond 5)
> (my-cond [#t #:whoops])
eval:10:0: my-cond: expected expression
  at: #:whoops
  in: (my-cond (#t #:whoops))
  parsing context:
   while parsing cond-clause
    term: (#t #:whoops)
    location: eval:10:0
```

In the implementation above, should we also annotate `more` to check that all of the arguments are clauses, instead of only checking the first clause at each step? That is:

```
; (my-cond CondClause ...) : Expr
(define-syntax my-cond
  (syntax-parser
    [(_)
```

```
     #'(void)]
    [(_ c:cond-clause more:cond-clause ...)
     #'(if c.condition
           c.result
           (my-cond more ...))]))
```

It can lead to earlier detection of syntax errors and better error messages, because the error is reported in terms of the original expression the user wrote, as opposed to one created by the macro for recursion. The cost is that the syntax-class check is performed again and again on later arguments; the number of `cond-clause` checks performed by this version is quadratic in the number of clauses it originally receives. One solution is to make the public `my-cond` macro check all of the clauses and then expand into a private recursive helper macro that only interprets one clause at a time.

```
; (my-cond CondClause ...) : Expr
(define-syntax my-cond
  (syntax-parser
    [(_ c:cond-clause ...)
     #'(my-cond* c ...)]))
; (my-cond* CondClause ...) : Expr
(define-syntax my-cond*
  (syntax-parser
    [(_)
     #'(void)]
    [(_ c:cond-clause more ...)
     #'(if c.condition
           c.result
           (my-cond* more ...))]))
```

This tension arises because a syntax class has two purposes: validation and interpretation. For a single term, validation should precede interpretation, but if a macro has many arguments (for example, a use of `my-cond` might have many `CondClause`s), how should we interleave validation and interpretation of the many terms? One appealing goal is to validate all arguments before interpreting any of them. Another appealing goal is to only "call" a syntax class once per term. Each goal constrains the ways we can define the syntax class and write the macro; achieving both goals is especially tricky.

A related question is this: How much of the task of interpeting a term belongs to the syntax class versus the macro that uses it? The division of responsibility between syntax class and macro affects the *interface* between them, and that interface affects how the macro is written. This question becomes more complicated when we add variants to a syntax class; we discuss the difficulties and solutions in detail in §6 "Enumerated Shapes".

5.2 Same Structure, Different Interpretation

Recall Exercise 9. The goal was to design a macro `my-evcase1` with the following shape:

```
;; (my-evcase1 Expr [Expr Expr] ...) : Expr
```

The exercise's description of the macro's behavior referred to "clauses", which is a hint that we should improve the specification by naming that argument shape. Let's do that now.

We already have a name for the shape `[Expr Expr]`; should we simply define the shape of `my-evcase1` in terms of `CondClause`? (Perhaps we should also generalize the name to `ClauseWith2Exprs` so it doesn't seem so tied to `my-cond`?)

No. The structure of the two shapes is the same, but the interpretation is different. Specifically, the first expression of a `CondClause` is treated as a condition, but the first expression of a `my-evcase1` clause is treated as a value for equality comparison. Furthermore, the two macros happen to have the same clause structure now, but if we add features to one or the other (and we will), they might evolve in different ways. In fact, they are likely to evolve in different ways *because* they have different interpretations.

So let's define a new shape, `EC1Clause`, for `my-evcase1` clauses:

```
;; EC1Clause ::= [Expr Expr]    -- comparison value, result
```

Here is the corresponding syntax class:

Now the macro has the shape

```
;; (my-evcase1 Expr EC1Clause ...) : Expr
```

One implementation strategy is to use `my-cond` as a helper macro. Here's a first attempt that isn't quite right:

```
(define-syntax my-evcase1
  (syntax-parser
    [(_ find:expr c:ec1-clause ...)
     #'(my-cond [(equal? find c.comparison) c.result] ...)]))
```

These examples illustrate the problem:

```
> (my-evcase1 (begin (printf "got a coin!\n") (* 5 5))
    [5 "nickel"]
    [10 "dime"]
    [25 "quarter"]
    [(/ 0) "infinite money!"])
```

```
got a coin!
got a coin!
got a coin!
"quarter"
> (define coins '(25 5 10 5))
> (define (get-coin) (begin0 (car coins) (set! coins (cdr coins))))
> (my-evcase1 (get-coin)
    [5 "nickel"]
    [10 "dime"]
    [25 "quarter"]
    [(/ 0) "infinite money!"])
/: division by zero
```

The initial expression is re-evaluated for every comparison, which is problematic if the expression has side-effects.

Here is a fixed implementation that uses a temporary variable to hold the value of the first expression:

```
(define-syntax my-evcase1
  (syntax-parser
    [(_ find:expr c:ec1-clause ...)
     #'(let ([tmp find])
         (my-cond [(equal? tmp c.comparison) c.result] ...))]))
```

Now the examples behave as expected:

```
> (my-evcase1 (begin (printf "got a coin!\n") (* 5 5))
    [5 "nickel"]
    [10 "dime"]
    [25 "quarter"]
    [(/ 0) "infinite money!"])
got a coin!
"quarter"
> (define coins '(25 5 10 5))
> (define (get-coin) (begin0 (car coins) (set! coins (cdr coins))))
> (my-evcase1 (get-coin)
    [5 "nickel"]
    [10 "dime"]
    [25 "quarter"]
    [(/ 0) "infinite money!"])
"quarter"
```

Exercise 10: Turn the examples above into test cases for `my-evcase1`. Check that the tests fail on the original version of the macro and succeed on the fixed version.

The `catch-output` macro from Exercise 3 and §3.3 “Proper Lexical Scoping, Part 2” is not quite enough to express these tests conveniently. Write a more general helper macro with the following shape:

```
;; (result+output Expr[R]) : Expr[(list R String)]
```

and use that to express your tests.

5.3 Helper Macros and Simple Expressions

Recall the implementation strategies for handling ellipsis shapes from §4.2 “Ellipses with Simple Shapes”. The first strategy was to write a recursive macro. Is it possible to implement `my-evcase1` using that strategy?

No. It is not possible to implement `my-evcase1` as a recursive macro, according to the shape we’ve given it, while guaranteeing that we evaluate the initial expression once. Compare this with fact that some list functions cannot be written purely as structural recursive functions. The `average` function is a good example: it can only be expressed by combining or adjusting the results of one or more structurally recursive helper functions.

We can, however, implement `my-evcase1` using a recursive helper macro. In fact, we’ve done that in the previous implementation, by using `my-cond`. But let’s try a different implementation using a recursive macro that has a shape that is similar to, although not identical to, that of `my-evcase1`. In particular, it is worth talking about the shape involved in the interface between the main, public macro and its private helper.

```
; (my-evcase1 Expr EC1Clause ...) : Expr
(define-syntax my-evcase1
  (syntax-parser
    [(_ find:expr c:ec1-clause ...)
     #'(let ([tmp find])
         (my-evcase1* tmp c ...))]))
```

So, what is the shape of the helper macro, `my-evcase1*`?

We could describe `my-evcase1*` with the following shape:

```
;; (my-evcase1* Expr EC1Clause ...) : Expr
```

but that’s the same shape as `my-evcase1`, so if we’re using the shapes to guide our design — specifically, our implementation options — we have not made any progress. The point of `my-evcase1*` is that its first argument is a simple variable reference, not any arbitrary expression whose evaluation might be costly or involve side effects. Let’s reflect that in the shape:

```
;; (my-evcase1* SimpleExpr EC1Clause ...) : Expr
```

The `SimpleExpr` shape is like `Expr`, except that it only contains expressions that we are willing to duplicate. That is, the expansion of the expression is simple and small, and the evaluation of the expression is trivial and does not involve side effects. Acceptable expressions include quoted constants and variable references. Usually, we also expect simple expressions to be constant, so a variable reference should be to a fresh local variable that is never mutated. Depending on the situation, there might be other expressions that we would accept as simple.

There is no separate syntax class for `SimpleExpr`; just use `expr` or omit the syntax class annotation. It is infeasible to *check* whether an expression is simple; instead, you should only make private macros accept `SimpleExpr` arguments, and you should check that all of the public macros that call them pass appropriate expressions.

In this example, let's assume that `my-evcase1*` is private and only `my-evcase` calls it. The initial expression that `my-evcase` gives to the helper is a local variable reference, which is simple.

Here is a recursive implementation of `my-evcase1*`:

```
; (my-evcase1* SimpleExpr EC1Clause ...) : Expr
(define-syntax my-evcase1*
  (syntax-parser
    [(_ tmp)
     #'(void)]
    [(_ tmp c:ec1-clause more ...)
     #'(if (equal? tmp c.comparison)
           c.result
           (my-evcase1* tmp more ...))]))
```

Note that the second clause duplicates the `tmp` argument.

6 Enumerated Shapes

This section introduces enumerated shapes — that is, shapes with multiple variants. An enumerated shape poses a problem in defining the interface between the corresponding syntax class and the macro that uses it. This section discusses several strategies for defining this interface.

6.1 Defining Enumerated Shapes

In the previous section, we extracted the definition of `CondClause` from the shape of the `my-cond` macro, and we defined a syntax class `cond-clause` corresponding to the shape.

Now let's extend `CondClause` with another *variant* that allows the clause's result to depend on the (true) value produced by the condition. This is similar to the `=>` clause form that Racket's `cond` macro supports, but we'll use a keyword, `#:apply`, to distinguish this form of clause for `my-cond`. Here is the updated shape definition:

```
;; CondClause ::=
;; | [Expr Expr]          -- represents condition, result
;; | [Expr #:apply Expr]  -- represents condition, function from condition to result
```

Here are some examples:

```
(define ls '((a 1) (b 2) (c 3)))
(my-cond [(assoc 'b ls) #:apply cadr] [#t 0])
; expect 2
(my-cond [(assoc 'z ls) #:apply cadr] [#t 0])
; expect 0
```

Here is one way the first example could expand (just the first step):

```
(let ([condition-value (assoc 'b ls)])
  (if condition-value
      (cadr condition-value)
      (my-cond [#t 0])))
```

We update the definition of `cond-clause` by adding another `pattern` clause for the new variant. Its second expression has a different intepretation, so we should use a different name for its pattern variable so that we don't confuse them:

```
(begin-for-syntax
  (define-syntax-class cond-clause
    #:attributes (condition result) ; !!
```

```
    (pattern [condition:expr result:expr])
    (pattern [condition:expr #:apply get-result:expr])))
```

There's a problem, though. The new pattern is fine by itself, but it doesn't fit with the existing `#:attributes` declaration. The second variant doesn't have a simple `result` expression; it interprets its second expression differently. The syntax class, though, needs a single interface that determines what nested attributes are bound when the syntax class is used in a macro and how the nested attributes are interpreted.

The interface between syntax class and macro is determined by how we allocate reponsibility between the two for the interpretation of the syntax class's terms. This problem is a fundamentally difficult one, and in general there is no single right answer, but there are some standard *interface strategies*:

- empty interface (redo case analysis)
- common meaning
- macro behavior
- code generation
- AST

Each has different tradeoffs, and some don't work in all situations. The following sections discuss each approach in greater detail.

6.1.1 Empty Interface Strategy (Redo Case Analysis)

One option is to give the syntax class no responsibility for interpreting its terms, and simply redo the case analysis in the macro. This is the *empty interface* strategy. The syntax class is still useful for input validation and as internal documentation, but since it performs no interpretation, we should declare that it exports no attributes.

```
(begin-for-syntax
  (define-syntax-class cond-clause
    #:attributes ()
    (pattern [condition:expr result:expr])
    (pattern [condition:expr #:apply get-result:expr])))
```

The following version of `my-cond` checks the syntactic structure of all of its arguments, then expands into a private recursive helper macro `my-cond*`, which performs the case analysis on each clause again:

```
; (my-cond CondClause ...) : Expr
(define-syntax my-cond
  (syntax-parser
    [(_ c:cond-clause ...)
     #'(my-cond* c ...)]))
; (my-cond* CondClause ...) : Expr
(define-syntax my-cond*
  (syntax-parser
    [(_)
     #'(void)]
    [(_ [condition:expr result:expr] more ...)
     #'(if condition
           result
           (my-cond* more ...))]
    [(_ [condition:expr #:apply get-result:expr] more ...)
     #'(let ([condition-value condition])
         (if condition-value
             (get-result condition-value)
             (my-cond* more ...)))]))
```

An advantage of this strategy is that it is nearly always a viable option. A disadvantage is that it duplicates syntax patterns, which introduces the possibility of discrepancies between the syntax class and the macro clauses. Such discrepancies can lead to problems that are difficult to catch and debug.

Exercise 11: Extend the definition of `CondClause` with one more variant as follows:

```
;; CondClause ::= ... | [Expr #:bind c:Id Expr{c}]
```

If the condition evaluates to a true value, it is bound to the given variable name and the result expression is evaluated in the scope of that variable. The scope of the variable does not include any other clauses.

Update the design of `cond-clause` and `my-cond` for the new `CondClause` variant using the strategy described in this section.

6.1.2 Common Meaning Interface Strategy

In some cases, it is possible to find a *common meaning* shared by all of the variants that is also sufficient for the macro to work with. In the case of `CondClause`, this is relatively straightforward: We can convert any "normal" clause into an "apply" clause by wrapping it in a function that ignores its argument. For example, instead of writing

```
(cond [(even? 2) 'e]
      [(odd? 2) 'o])
```

we could instead write

```
(cond [(even? 2) #:apply (lambda (ignore) 'e)]
      [(odd? 2) #:apply (lambda (ignore) 'o)])
```

That is, the second clause form is strictly more general than the first clause form. We don't need to actually rewrite the whole clauses; instead, we can change the attributes of `cond-clause` to `condition` and `get-result` to represent the second form, and we can change the first form to *compute* the `get-result` attribute using `#:with`. Here is the new syntax class:

```
(begin-for-syntax
  (define-syntax-class cond-clause
    #:attributes (condition    ; Expr[(U #f C)]
                  get-result)  ; Expr[C -> Any]
    (pattern [condition:expr result:expr]
             #:with get-result #'(lambda (ignore) result))
    (pattern [condition:expr #:apply get-result:expr])))
```

I've also added comments with shape annotations for the attributes, to help me remember their intended interpretation.

Here is an implementation of `my-cond` using this version of `cond-clause` and its attributes:

```
; (my-cond CondClause ...) : Expr
(define-syntax my-cond
  (syntax-parser
    [(_)
     #'(void)]
    [(_ c:cond-clause more ...)
     #'(let ([condition-value c.condition])
         (if condition-value
             (c.get-result condition-value)
             (my-cond more ...)))]))
```

You might worry that introducing a `lambda` wrapper and a function call for every simple clause form will make the generated code run slower. After all, `lambda` requires a closure allocation, right? In this case, that is not true. The generated `lambda` wrappers appear directly in application position, and the Racket compiler is more than smart enough to inline those applications away. So even though the new version of the macro expands to different Racket code for simple clauses, the compiler produces exactly the same compiled code, with zero run-time overhead.

Exercise 12: See Exercise 11 for a revised definition of `CondClause`. Update the design of `cond-clause` and `my-cond` for the new `CondClause` variant using the strategy described in this section.

6.1.3 Macro Behavior Interface Strategy

If it is difficult to find a common interface for all of a syntax class's variants based solely on their contents, an alternative is to design the interface based on the *macro behavior*. This is similar to shifting from "functional" style operations defined separately from a data type to "object-oriented" style methods where behavior is defined together with the type and its variants. The potential downside, of course, is that it couples the syntax class more tightly with the macro.

In this example, we can move the responsibility for testing the condition and producing the result if the clause is selected from the macro to the syntax class. What is left, then? If the clause is not selected (that is, the clause "fails"), it needs to be told how to continue the search for an answer. We can represent "how to continue" with a thunk of type `(-> Any)`; this kind of thunk is traditionally called a "failure continuation". (This sense of the word "continuation" does not refer to the kind of value exposed by `call/cc`, etc.) So the whole clause is represented by a function that takes a failure continuation and produces an answer; it has the type `(-> Any) -> Any`. We define `cond-clause` with a single attribute, `code`, containing an expression for that function:

```
(begin-for-syntax
  (define-syntax-class cond-clause
    #:attributes (code) ; Expr[(-> Any) -> Any]
    (pattern [condition:expr result:expr]
             #:with code #'(lambda (fail)
                             (if condition result (fail))))
    (pattern [condition:expr #:apply get-result:expr]
             #:with code #'(lambda (fail)
                             (let ([condition-value condition])
                               (if condition-value
                                   (get-result condition-value)
                                   (fail)))))))
```

Now in the recursive case, the `my-cond` macro just calls the clause's `code` with a failure continuation that tries the rest of the clauses:

```
; (my-cond CondClause ...) : Expr
(define-syntax my-cond
  (syntax-parser
    [(_)
     #'(void)]
    [(_ c:cond-clause more ...)
     #'(c.code (lambda () (my-cond more ...)))]))
```

Again, you might worry that the use of `lambda` leads to run-time inefficiency, but the way this macro uses `lambda` is easily optimized away by the compiler.

Exercise 13: See Exercise 11 for a revised definition of `CondClause`. Update the design of `cond-clause` and `my-cond` for the new `CondClause` variant using the strategy described in this section.

6.1.4 Code Generator Interface Strategy

Suppose, though, that we really wanted to produce more natural looking code, perhaps for readability. Here's a variation on the previous solution: Instead of exporting a syntax-valued attribute that takes a run-time failure continuation, export a *function-valued* attribute that takes a compile-time failure *expression* and produces an expression implementing `my-cond`'s behavior for that clause. That is, the attribute represents a *code generator* for the clause. For example:

```
(begin-for-syntax
  (define-syntax-class cond-clause
    #:attributes (make-code) ; Syntax[Expr] -> Syntax[Expr]
    (pattern [condition:expr result:expr]
             #:attr make-code (lambda (fail-expr)
                                #`(if condition result #,fail-
expr)))
    (pattern [condition:expr #:apply get-result:expr]
             #:attr make-code (lambda (fail-expr)
                                #`(let ([condition-
value condition])
                                    (if condition-value
                                        (get-result condition-
value)

                                        #,fail-expr))))))
```

Note that the `make-code` attribute is declared with a type, not a shape, and it is defined using `#:attr` instead of `#:with`.

Here is the corresponding definition of `my-cond`:

```
; (my-cond CondClause ...) : Expr
(define-syntax my-cond
  (syntax-parser
    [(_)
     #'(void)]
    [(_ c:cond-clause more ...)
     ((datum c.make-code) #'(my-cond more ...))]))
```

The value of `c.make-code` is not syntax, so we cannot use it in a syntax template. We use `datum` to get the attribute value (a function), and apply it to syntax representing an expression handling the rest of the clauses.

Here's another version of the macro that checks all of the clauses first and then uses foldr to process all of their code generators:

```
; (my-cond CondClause ...) : Expr
(define-syntax my-cond
  (syntax-parser
    [(_ c:cond-clause ...)
     (foldr (lambda (make-code rec-expr)
              (make-code rec-expr))
            #'(void)
            (datum (c.make-code ...)))]))
```

The expression `(datum (c.make-code ...))` has type `(Listof (Syntax[Expr] -> Syntax[Expr]))`.

Exercise 14: See Exercise 11 for a revised definition of `CondClause`. Update the design of `cond-clause` and `my-cond` for the new `CondClause` variant using the strategy described in this section.

6.1.5 AST Interface Strategy

The *AST strategy* is a variation on the §6.1.1 "Empty Interface Strategy (Redo Case Analysis)" approach, which has the macro redo the syntax class's case analysis. But in this variation, instead of the macro doing case analysis on the syntax, the syntax class parses its terms into values in some AST datatype, and then the macro does case analysis on the AST values.

This results in a larger interface between the syntax class and the macro or macros that use it, because the interface includes the AST datatype definition. On the other hand, the syntax class does not have to specialize its interpretation based on the behavior of any specific macro. Furthermore, the case analysis performed by the macro can be simpler and faster, and errors will be easier to catch and debug, compared to discrepancies between syntax class and macro syntax patterns.

Here is a definition of an AST datatype (`ClauseRep`) and a version of the `cond-clause` syntax class that exports a single `ast` attribute containing a `ClauseRep` value:

```
(begin-for-syntax
  ; A ClauseRep is an instance of one of the following:
  (struct clause:normal
    (condition    ; Syntax[Expr]
     result))     ; Syntax[Expr]
  (struct clause:apply
    (condition    ; Syntax[Expr[(U #f C)]
     get-result)) ; Syntax[Expr[C -> Any]]
```

```
(define-syntax-class cond-clause
  #:attributes (ast) ; ClauseRep
  (pattern [condition:expr result:expr]
           #:attr ast (clause:normal #'condition #'result))
  (pattern [condition:expr #:apply get-result:expr]
           #:attr ast (clause:apply #'condition #'get-result))))
```

The type `ClauseRep` represents parsed terms of the shape `CondClause`. I've given them separate names here for clarify, but in practice I often use the same name for type and shape. The context usually disambiguates them.

Here is one implementation of the `my-cond` macro, using `match` to do case analysis on the clause ASTs:

```
;; (my-cond CondClause ...) : Expr
(define-syntax my-cond
  (syntax-parser
    [(_ c:cond-clause ...)
     (foldr (lambda (ast rec-code)
              (match ast
                [(clause:normal condition result)
                 #`(if #,condition #,result #,rec-code)]
                [(clause:apply condition get-result)
                 #`(let ([condition-value #,condition])
                     (if condition-value (#,get-result condition-
value) #,rec-code))]))
            #'(void)
            (datum (c.ast ...)))]))
```

Exercise 15: See Exercise 11 for a revised definition of `CondClause`. Update the design of `cond-clause` and `my-cond` for the new `CondClause` variant using the strategy described in this section. Double-check that a `#:bind`-clause variable is not visible in subsequent clauses!

6.2 Designing Enumerated Syntax

When you design an enumerated syntax shape, you must avoid ambiguity; or if you cannot completely avoid it, you must manage it carefully. To elaborate, let's consider some alternative syntaxes for cond-clauses.

For `AltCondClauseV1`, let's generalize the simple form, so that the result is determined not by a single expression but by one or more body terms. And let's indicate the second form with the identifier `apply` instead of the keyword `#:apply`.

```
;; AltCondClauseV1 ::=
;; | [Expr Body ...+]
;; | [Expr apply Expr]
```

This syntax design is *bad*, because there are two variants with different meanings that contain the same terms. In fact, every term that matches the second variant also matches the first.

One could argue that a programmer is unlikely to simply write `apply` at the beginning of a result body intending it to be evaluated as an expression. It would have no effect; its presence would be completely useless. Still, programmers regularly trip on "out of the way" inconsistencies, and it's a better habit to keep comfortable safety margins.

Let's change the definition slightly so that instead of `apply`, we use the identifier `=>` to indicate the second clause form:

```
;; AltCondClauseV2 ::=
;; | [Expr Body ...+]
;; | [Expr => Expr]
```

This syntax design is okay; it is in fact the design Racket uses for `cond` clauses. There are two crucial details, though. First, the `=>` variant must be recognized not by the symbolic content of the `=>` identifier, but by checking whether it is a reference to Racket's `=>` binding. Second. Racket defines `=>` as a macro that always signals a syntax error. So even though we can interpret `=>` as an expression, it is never a *valid* expression. In practice, we only care about avoiding overlaps with the set of valid expressions, so `AltCondClauseV2` is okay.

Both properties are needed to avoid ambiguity. If we checked for `=>` as a symbol, then a programmer could define a local variable (or macro) named `=>`, which would then be a valid expression, so there would be overlap. And if `=>` were a valid expression, that also creates an overlap. (That's the problem with the `apply` variant in `AltCondClauseV1`.)

Finally, although the shape is okay, we must be careful when writing the corresponding syntax patterns. First, we must declare `=>` as a *literal*; otherwise it would be treated as another pattern variable. Here is a first draft of the corresponding syntax class definition, based on the "macro behavior interface" strategy:

```
(begin-for-syntax
  (define-syntax-class alt-cond-clause-v2
    #:attributes (code) ; Expr[(-> Any) -> Any]
    #:literals (=>)
    (pattern [condition:expr result:expr ...+]
             #:with code ····)
    (pattern [condition:expr => get-result:expr]
             #:with code ····)))
```

The problem is that a clause term like `[a => b]` matches the first pattern, and so the syntax class interprets it as a simple condition and result-body clause. That is, the same issue we dealt with earlier at the shape level shows up again at the syntax pattern level. It shows up again because even though => cannot be a valid expression, the expr syntax class doesn't know that. This issue is not specific to syntax classes; it would also come up if we did the case analysis in the macro patterns.

The solution involves two steps. First, reorder the patterns to put the => pattern first. Second, use ~! ("cut") to commit to the first pattern after seeing =>. Here is the code:

```
(begin-for-syntax
  (define-syntax-class alt-cond-clause-v2
    #:attributes (code) ; Expr[(-> Any) -> Any]
    #:literals (=>)
    (pattern [condition:expr => ~! get-result:expr]
             #:with code ····)
    (pattern [condition:expr result:expr ...+]
             #:with code ····)))
```

Without the ~!, a term like [a => b c] would be considered a valid instance of the second pattern, rather than an invalid instance of the first pattern. (Try it and see what happens!)

The complexity of overlaps with expressions is one reason that keywords were introduced into Racket. Since both the Expr shape and the expr syntax class consider themselves completely disjoint from keywords, they avoid these issues completely. (A related issue does emerge when dealing with partly-expanded code, distinguishing definitions from expressions, for example. We'll talk about that later. FIXME-REF)

7 Multi-Term Shapes

This section introduces "multi-term" shapes, used to describe syntactic elements like keyword arguments.

7.1 Shapes for Multiple Terms

In Racket, the syntax of a "keyword argument" to a function does not consist of a single term; it consists of two terms, a keyword followed by the argument term. That is, the logical grouping structure does not correspond with the term structure. The syntax of macros generally follows the same idiom: a macro keyword argument consists of the keyword and zero or more argument terms, depending on the keyword. For example, the `#:attributes` keyword used by `define-syntax-class` takes one argument (a list of attributes); and the `#:with` keyword takes two (a syntax pattern and an expression).

We can define shapes that stand for multiple terms like this:

```
;; AttributesClause ::= #:attributes (Id ...)
;; WithClause       ::= #:with SyntaxPattern Expr
```

Multi-term shapes are represented by *splicing syntax classes*, which encapsulate *head syntax patterns* (so called because they match some variable-length "head" of the list term).

Let's extend `my-cond` with support for a `#:do` clause that has a single `Body` argument. That will allow us to include definitions between tests. Here's an example:

```
(define ls '((a 1) (b 2) (c 3)))
(define entry (assoc 'b ls))
(my-cond [(not entry) (error "not found")]
         #:do (define value (cadr entry))
         [(even? value) 'even]
         [(odd? value) 'odd])
```

Here is the revised definition of `CondClause`, which is now a multi-term shape:

```
;; CondClause ::=
;; | [Expr Expr]
;; | [Expr #:apply Expr]
;; | #:do Body
```

Here is the corresponding syntax class, including only the patterns:

```
(begin-for-syntax
```

```
(define-splicing-syntax-class cond-clause
  #:attributes ()
  (pattern [condition:expr result:expr])
  (pattern [condition:expr #:apply get-result:expr])
  (pattern (~seq #:do body:expr))))
```

We must declare my-cond using define-splicing-syntax-class, and we must use ~seq to wrap multiple-term patterns.

What interface can we give to the syntax class, and how do we implement the macro? Let's review the implementations of my-cond from §6.1 "Defining Enumerated Shapes":

- The approach of redoing the case analysis from §6.1.1 "Empty Interface Strategy (Redo Case Analysis)" would also still work.
- The approach from §6.1.2 "Common Meaning Interface Strategy" no longer works, because #:do clauses are not a special case of #:apply clauses.
- The failure-continuation approach from §6.1.3 "Macro Behavior Interface Strategy" no longer works, because the scope of definitions within #:do clauses should cover the rest of the clauses, but the failure continuation is received as a closure value, and there's no way to affect its environment.
- The code generator approach from §6.1.4 "Code Generator Interface Strategy" would still work, since the code generator for the #:do clause can put the expression representing the rest of the clauses in the scope of the #:do-clause's definitions.
- The AST approach from §6.1.5 "AST Interface Strategy" would still work. We would need to update the AST datatype with a new variant and update the macro's case analysis to handle it.

This is a good summary of how robust each of these strategies is to changes in the shape.

7.1.1 Redo Case Analysis

For the empty interface, we simply add a case to the private, recursive macro:

```
(begin-for-syntax
  (define-splicing-syntax-class cond-clause
    #:attributes ()
    (pattern [condition:expr result:expr])
    (pattern [condition:expr #:apply get-result:expr])
    (pattern (~seq #:do body:expr))))
; (my-cond CondClause ...) : Expr
(define-syntax my-cond
```

```
  (syntax-parser
    [(_ c:cond-clause ...)
     #'(my-cond* c ...)]))
; (my-cond* CondClause ...) : Expr
(define-syntax my-cond*
  (syntax-parser
    [(_)
     #'(void)]
    [(_ [condition:expr result:expr] more ...)
     #'(if condition
           result
           (my-cond* more ...))]
    [(_ [condition:expr #:apply get-result:expr] more ...)
     #'(let ([condition-value condition])
         (if condition-value
             (get-result condition-value)
             (my-cond* more ...)))]
    [(_ #:do body:expr more ...)
     #'(let ()
         body
         (my-cond* more ...))]))
```

7.1.2 Code Generator

With the code generator strategy, the new implementation simply involves two changes to the old implementation. We must change `define-syntax-class` to `define-splicing-syntax-class`, and we must add the third variant as below. The definition of `my-clause` itself does not change.

```
(begin-for-syntax
  (define-splicing-syntax-class cond-clause
    #:attributes (make-code) ; Syntax[Expr] -> Syntax[Expr]
    (pattern [condition:expr result:expr]
             #:attr make-code (lambda (fail-expr)
                                #`(if condition result #,fail-
expr)))
    (pattern [condition:expr #:apply get-result:expr]
             #:attr make-code (lambda (fail-expr)
                                #`(let ([condition-
value condition])
                                    (if condition-value
                                        (get-result condition-
value)
                                        #,fail-expr))))
    (pattern (~seq #:do body:expr)
```

```
              #:attr make-code (lambda (fail-expr)
                                 #`(let ()
                                     body
                                     #,fail-expr)))))
; (my-cond CondClause ...) : Expr
(define-syntax my-cond
  (syntax-parser
    [(_ c:cond-clause ...)
     (foldr (lambda (make-code rec-expr)
              (make-code rec-expr))
            #'(void)
            (datum (c.make-code ...)))]))
```

7.1.3 AST

Exercise 16: Adapt the solution from §6.1.5 "AST Interface Strategy" to support `#:do`-clauses.

7.2 Optional Shapes

A common kind of multi-term shape is one that has two (or more variants), one of which consists of zero terms. A good naming convention for such shapes and syntax classes is to start them with "maybe" or "optional". For example, we could add an optional final `#:else` clause to `my-cond`, like this:

```
;; (my-cond CondClause ... MaybeFinalCondClause) : Expr
```

where `MaybeFinalCondClause` is defined as follows:

```
;; MaybeFinalCondClause ::= ε | #:else Expr
```

Here I've used ε to represent zero terms.

The corresponding syntax class for `MaybeFinalCondClause` must be a splicing syntax class. The interpretation of the possible final clause is that it provides a condition-free result if none of the previous clauses were selected; if absent, the result is `(void)`. So we can represent the interpretation with a single attribute holding an expression:

```
(begin-for-syntax
  (define-splicing-syntax-class maybe-final-cond-clause
    #:attributes (result) ; Expr
    (pattern (~seq)
             #:with result #'(void))
    (pattern (~seq #:else result:expr))))
```

Here is the macro, starting from the code-generation implementation above. The only changes are to the pattern and the use of `#'fc.result` instead of `#'(void)` in the call to `foldr`.

```
(define-syntax my-cond
  (syntax-parser
    [(_ c:cond-clause ... fc:maybe-final-cond-clause)
     (foldr (lambda (make-code rec-expr)
              (make-code rec-expr))
            #'fc.result
            (datum (c.make-code ...)))]))
```

7.3 Shapes, Types, and Scopes (⋆)

In §3.4 “The Identifier Shape” and §3.5 “Expressions, Types, and Contracts” we explored how to express scoping and type relationships between parts of a shape. Can we extend the notation to express the scoping of the `#:do` form of `CondClause`?

Recall the example program:

```
(define ls '((a 1) (b 2) (c 3)))  ; (Listof (list Symbol Integer))
(define entry (assoc 'b ls))      ; (list Symbol Integer)
(my-cond [(not entry) (error "not found")]
         #:do (define value (cadr entry))
         [(even? value) 'even]
         [(odd? value) 'odd])
```

How does each clause affect the environment of subsequent clauses? The first clause has no effect on the environments of the following clauses. The second clause adds a variable binding `value` with type `Integer`. More generally, since we could define multiple variables using `define-values` or combine definitions with `begin`, each clause might bind a *set* of variables, and each variable has a corresponding type. The first clause produces no bindings (so $\varnothing$, the empty set); the second set produces `{value:Integer}`; the third and fourth clauses also produce $\varnothing$. Let’s add a parameter to `CondClause` representing the bindings it “produces” — that is, the bindings it adds to the environments of subsequent clauses. We need to change the way we write the shape definition:

```
;; CondClause[∅] ::= [Expr Expr]
;; CondClause[∅] ::= [Expr #:apply Expr]
;; CondClause[Δ] ::= #:do Body[Δ]
```

We need the same information from the `Body` shape. Note that Δ does not stand for a type; it stands for a set of pairs of names and types — that is, a fragment of a type environment.

In the second clause of this example, Δ is `{value:Integer}`. That is:

```
#:do (define value (cadr entry))    : CondClause[{value:Integer}]
```

because

```
(define value (cadr entry))         : Body[{value:Integer}]
```

We also need to change the way we talk about lists of clauses. The notation `CondClause ...` doesn't give us a good way to talk about the relationship between different clauses. Instead, let's define a multi-term shape called `CondClauses`:

```
;; CondClauses ::= ε
;; CondClauses ::= CondClause[Δ] CondClauses{Δ}
```

By `CondClauses{Δ}` I mean that all expressions, body terms, etc within the clause are in the scope of the additional bindings described by Δ. That is, an environment annotation is implicitly propagated to all of a shape's sub-shapes. The second line says that in `CondClauses` sequence, if one clause produces some bindings, then subsequent clauses are in their scope.

Here are the shape definitions with the environment annotations made fully explicit, where Γ stands for a type environment:

```
;; CondClause{Γ}[∅] ::= [Expr{Γ} Expr{Γ}]
;; CondClause{Γ}[∅] ::= [Expr{Γ} #:apply Expr{Γ}]
;; CondClause{Γ}[Δ] ::= #:do Body{Γ}[Δ]

;; CondClauses{Γ} ::= ε
;; CondClauses{Γ} ::= CondClause[Δ] CondClauses{Γ,Δ}
```

Finally, here is the shape of `my-cond`:

```
;; (my-cond CondClauses)    : Expr      -- implicit
;; (my-cond CondClauses{Γ}) : Expr{Γ}   -- explicit
```

That shows how to use the shape notation to specify type and scoping relationships between components of shapes.

For many macros, it is probably unnecessary to put this level of detail in the shape declarations. A more practical approach might be to limit the shapes to specifying syntactic structure and interpretation, as we've been doing, and describe the scoping of shapes and macros in prose.

On the other hand, a more precise specification is sometimes useful when writing macros with complicated binding structures. So keep this tool in mind in case you need it.

8 Recursive Shapes

8.1 The Datum Shape

The Datum shape contains all number terms, identifier terms, and other atomic terms, as well as all list, vector, hash, box, and prefab struct terms containing Datum elements. That is, Datum contains any term the corresponds to a readable value.

The Datum shape represents the intention to use the term as a literal within a quote expression, or to convert it to a compile-time value using syntax->datum.

There is no syntax class corresponding to Datum.

Let's design the macro my-case1, which is like my-evcase1 from Exercise 9 and §5.2 "Same Structure, Different Interpretation" except that each clause's comparison value is given as a datum rather than an expression. That is, the macro's shape is:

```
;; (my-case1 Expr [Datum Expr] ...) : Expr
```

Here is an example:

```
(my-case1 (begin (printf "got a coin!\n") (* 5 5))
  [5 "nickel"] [10 "dime"] [25 "quarter"])
; expect print once, "quarter"
```

Here is an implementation:

```
; (my-case1 Expr [Datum Expr] ...) : Expr
(define-syntax my-case1
  (syntax-parser
    [(_ to-match:expr [datum result:expr] ...)
     #'(let ([tmp to-match])
         (cond [(equal? tmp (quote datum)) result] ...))]))
```

I often spell out quote in a syntax template when it is applied to a term containing pattern variables, to remind myself that the quoted "constant" can vary based on the macro's arguments.

Here is another implementation:

```
; (my-case1 Expr [Datum Expr] ...) : Expr
(define-syntax my-case1
  (syntax-parser
    [(_ to-match:expr [datum result:expr] ...)
     #'(let ([tmp to-match])
         (cond [(equal? tmp datum) result] ...))]))
```

This implementation is *wrong*, because the Datum arguments are not used within a quote expression. Never implicitly treat a Datum as an Expr! One obvious problem is that not every datum is self-quoting. The following example should return "matched", but it raises an error instead:

```
> (my-case1 (list 123)
    [(123) "matched"])
application: not a procedure;
 expected a procedure that can be applied to arguments
  given: 123
```

Even a datum that is normally self-quoting can carry a lexical context with an alternative #%datum binding that gives it some other behavior. For example:

```
> (let-syntax ([#%datum (lambda (stx) (raise-syntax-error #f "no
self-quoting!" stx))])
    (my-case1 '2 [1 'one] [2 'two] [3 'lots]))
eval:7:0: #%datum: no self-quoting!
  in: (#%datum . 1)
```

This particular example is admittedly uncommon. A more common problem is that the datum is computed by a macro, and depending on how it is coerced to a syntax object it may or may not get a lexical context with Racket's #%datum binding. Avoid all of these problems by treating Datum and Expr as distinct shapes. If you have a datum and you want an expression that evaluates to the same value at run time, put the datum in a quote expression.

Exercise 17: Generalize my-case1 to my-case, which has a list of datums in each clause. That is, the macro has the following shape:

```
;; (my-case Expr [(Datum ...) Expr] ...) : Expr
```

8.2 Datum as a Recursive Shape

Could we write a definition of Datum rather than treating it as a basic (that is, primitive) shape? The full definition would be quite complicated, since Racket has many kinds of readable values, and it occasionally adds new ones. Let's simplify the question to datum terms built from identifiers, numbers, booleans, strings, and proper lists; let's call this shape SimpleDatum. We can define it as a *recursive shape* as follows:

```
;; SimpleDatum ::= (SimpleDatum ...) | SimpleAtom
;; SimpleAtom  ::= Identifier | Number | Boolean | String
```

I have collected the base cases into a separate shape, SimpleAtom, for convenience.

Like the corresponding shapes, the simple-datum syntax class is recursive; the simple-atom syntax class is not. Let's discuss simple-atom first.

The simple-atom syntax class presents a challenge: There is a built-in syntax class for identifiers (id), but how do we check whether a term contains a number, a boolean, or a string? Given a syntax object, we can extract the corresponding plain Racket value by calling syntax->datum. Then we can use the ordinary number?, boolean?, and string? predicates. An identifier is just a syntax object containing a symbol, so we can cover the identifier case by adding a symbol? predicate check to the others. Finally, we perform this check using a ~fail pattern; if the check fails, then the syntax class does not accept the term. Here is the definition:

```
(begin-for-syntax
  ; simple-atom? : Any -> Boolean
  (define (simple-atom? v)
    (or (symbol? v) (number? v) (boolean? v) (string? v)))
  (define-syntax-class simple-atom
    #:attributes ()
    ; (pattern a #:when (simple-atom? (syntax->datum #'a)))
    (pattern a #:and (~fail #:unless (simple-atom? (syntax-
>datum #'a))))))
```

In most cases, it is better to use #:when to perform checks like this, but this is one of the few cases where we don't want a "post-traversal" check that dominates other matching failures. The difference between the two only affects the way errors are reported.

The simple-datum syntax class is straightforward. The recursive case in the shape simply translates to a syntax pattern with recursive syntax class annotations:

```
(begin-for-syntax
  (define-syntax-class simple-datum
    #:attributes ()
    (pattern (elem:simple-datum ...))
    (pattern atom:simple-atom)))
```

There are no attributes, because the only interpretation that SimpleDatum supports can be achieved with quote or syntax->datum on the term itself.

8.3 Quasiquotation

Let's define my-quasiquote, a simple version of Racket's quasiquote macro. Its argument has a shape like SimpleDatum, except that it can have "escapes" to Racket expressions so we can compute values to insert into the result. The shape of the macro is the following:

```
;; (my-quasiquote QuasiDatum) : Expr
```

where QuasiDatum is defined as follows:

```
;; QuasiDatum ::= (escape Expr)
;;              | (QuasiDatum ...)
;;              | SimpleAtom
```

What does escape mean in this shape definition? That is, how do we recognize the escape form of QuasiDatum. There are two possibilities: we could recognize escape either as a *symbolic literal* or as a *reference literal*. For this example, we'll recognize escape by reference; we'll show an example of symbolic literals later in FIXME-REF.

Recognizing escape as a reference means that there must be a binding of escape for the reference to refer to. Since we don't intend escape to have any meaning as a Racket expression, we should define it as a macro that always raises a syntax error:

```
(define-syntax escape
  (lambda (stx) (raise-syntax-error #f "illegal use of es-
cape" stx)))
```

The error only occurs if the macro is expanded by the Racket macro expander; our macros and syntax classes can still recognize references to it without triggering the error. Note that the module that provides my-quasiquote must also provide escape so that users of my-quasiquote can refer to this escape binding.

Now we can implement the quasi-datum syntax class. We declare escape as a (reference) literal using #:literals; then occurrences of escape in the syntax patterns are treated as literals instead of as pattern variables. Here is the definition, without attributes:

```
(begin-for-syntax
  (define-syntax-class quasi-datum
    #:literals (escape)
    (pattern (escape code:expr))
    (pattern (elem:quasi-datum ...))
    (pattern a:simple-atom)))
```

What interface should we give to the quasi-datum syntax class? Recall the interface strategies from §6.1 "Defining Enumerated Shapes". Most of them are applicable here; with the possible exception of the common meaning approach. Let's use the macro behavior strategy. The my-quasiquote interprets QuasiDatum as instructions to construct a value from a mixture of constants and values computed by escaped Racket expressions. We can represent that with a syntax attribute, code, containing an expression that produces the QuasiDatum's value. Here is the definition:

```
(begin-for-syntax
  (define-syntax-class quasi-datum
```

```
  #:attributes (code) ; Expr
  #:literals (escape)
  (pattern (escape code:expr))
  (pattern (elem:quasi-datum ...)
           #:with code #'(list elem.code ...))
  (pattern a:simple-atom
           #:with code #'(quote a))))
```

That is, the escape form contains the necessary expression directly; the list form constructs a list from the values constructed by its components; and an atom is interpreted as a value by quoting it.

The macro simply expands into its argument's code expression:

```
(define-syntax my-quasiquote
  (syntax-parser
    [(_ qd:quasi-datum)
     #'qd.code]))
```

Here are some examples:

```
> (my-quasiquote (1 2 () abc xyz))
'(1 2 () abc xyz)
> (my-quasiquote (1 2 (escape (+ 1 2))))
'(1 2 3)
> (my-quasiquote ((expression (+ 1 2)) (value (escape (+ 1 2)))))
'((expression (+ 1 2)) (value 3))
> (my-quasiquote (a (b (c (d (e (f (escape (string-
>symbol "g")))))))))
'(a (b (c (d (e (f g))))))
```

Because escape is recognized by reference, it can be made unavailable by shadowing, or it can be aliased to another name:

```
> (let ([escape 'piña-colada])
    (my-quasiquote (1 2 (escape (+ 1 2)))))
'(1 2 (escape (+ 1 2)))
> (let-syntax ([houdini (make-rename-transformer #'escape)])
    (my-quasiquote ((houdini 'jacket) (houdini 'water-tank))))
'(jacket water-tank)
```

The behavior of this example is questionable, though:

```
> (my-quasiquote (1 2 (escape 3 4)))
'(1 2 (escape 3 4))
```

Do we want (escape 3 4) to be interpreted as a three-element list, or do we want to consider it an ill-formed escape form and report an error? I prefer to consider it an error. That is, whenever the quasi-datum parser gets a term that starts (escape ␣ ...), it should *commit* to parsing it according to the escape pattern. This problem is similar to the problem with => clauses in §6.2 "Designing Enumerated Syntax". The solution is the same too: We must put the escape pattern before any other pattern it might overlap with, and we must add a cut (~!) immediately after the escape literal. Here is the updated code:

```
(begin-for-syntax
  (define-syntax-class quasi-datum
    #:attributes (code) ; Expr
    #:literals (escape)
    (pattern (escape ~! code:expr))
    (pattern (elem:quasi-datum ...)
             #:with code #'(list elem.code ...))
    (pattern a:simple-atom
             #:with code #'(quote a))))
```

Now the example signals an error instead:

```
> (my-quasiquote (1 2 (escape 3 4)))
eval:25:0: my-quasiquote: unexpected term
  at: 4
  in: (my-quasiquote (1 2 (escape 3 4)))
  parsing context:
   while parsing quasi-datum
    term: (escape 3 4)
    location: eval:25:0
   while parsing quasi-datum
    term: (1 2 (escape 3 4))
    location: eval:25:0
```

There's one remaining issue with this implementation. Consider the following example:

```
> (my-quasiquote (1 2 3 4 5))
'(1 2 3 4 5)
```

This example has no escapes, so its result could be implemented with a simple quote expression. But this is how the macro expands:

```
(my-quasiquote (1 2 3 4 5))
⇒
(list (quote 1) (quote 2) (quote 3) (quote 4) (quote 5))
```

Let's optimize my-quasiquote so that it uses quote at the highest levels possible. Here's one strategy: we add an boolean-valued const? attribute that is true when a term has no escapes. A list `QuasiDatum` is constant if all of its elements are constant; in that case, its code can be simply the quotation of the elements. Here is the updated syntax class:

```
(begin-for-syntax
  (define-syntax-class quasi-datum
    #:attributes (const? ; Boolean
                  code)  ; Expr
    #:literals (escape)
    (pattern (escape code:expr)
             #:attr const? #f)
    (pattern (elem:quasi-datum ...)
             #:attr const? (andmap values (datum (elem.const? ...)))
             #:with code (if (datum const?)
                             #'(quote (elem ...))
                             #'(list elem.code ...)))
    (pattern a:simple-atom
             #:attr const? #t
             #:with code #'(quote a))))
```

Exercise 18: Implement `quasi-datum` and `my-quasiquote` (the unoptimized version) according to the empty interface strategy and a recursive `my-quasiquote` macro. Is it possible to implement the `quote` optimization using this approach?

Exercise 19: Extend the definition of `QuasiDatum` as follows:

```
;; QuasiDatum ::= ... | (cellophane QuasiDatum)
```

A `cellophane` wrapper is simply discarded from the constructed value; that is, the `QuasiDatum` `(cellophane` *`qd`*`)` is equivalent to *`qd`*. For example:

```
(my-quasiquote (1 2 (cellophane 3) (escape (* 2 2))))
; expect '(1 2 3 4)
(my-quasiquote (1 (cellophane 2) (cellophane (cellophane 3))))
; expect '(1 2 3)
```

Start with the unoptimized version of the `quasi-datum` syntax class using the macro behavior strategy. After you have updated (and tested) that version, implement a similar optimization for the updated `QuasiDatum` shape. For example, the second example above should expand directly to `(quote (1 2 3))`. (Hint: What assumption made by the original optimization does the updated shape violate?)

9 Compile-Time Computation and Information

The section discusses macros that do computation at compile time, and it introduces a shape for compile-time information bound to an identifier.

9.1 Compile-Time Computation

This example is based on the `scramble/regexp` library. The `parser-tools/lex` library implements a similar notation.

I hate writing regular expressions. At least, I hate writing them once they get over twenty characters long, or have more than two "report" groups, or have character classes involving special characters, or....

Let's design a macro that takes a pleasant, compositional S-expression notation and automatically translates it at compile time to a regular expression literal — specifically, a `pregexp` literal.

9.1.1 Computation, Whenever

Wait! Why make this a macro? I can define an ordinary run-time AST datatype (call it `RE`) for representing regular expressions, and I can write a function that translates an `RE` to a `pregexp` string.

Here is the `RE` type. For simplicity, it only represents handles a subset of §4.8.1 "Regexp Syntax".

```
;; An RE is one of
;; - (re:or (NonemptyListof RE))
;; - (re:cat (Listof RE))
;; - (re:repeat Boolean RE)
;; - (re:report RE)
;; - (re:chars (NonemptyListof Range))
;; A Range is (rng Char Char)

(struct re:or (res) #:prefab)
(struct re:cat (res) #:prefab)
(struct re:repeat (plus? re) #:prefab)
(struct re:report (re) #:prefab)
(struct re:chars (ranges) #:prefab)
(struct rng (lo hi) #:prefab)
```

I'll explain why I use `#:prefab` structs in a later section.

Here is the code to translate an `RE` value into a `pregexp`-compatible string. The functions are organized according to what nonterminal in the ⟨*regexp*⟩ grammar they produce, to handle the precedence of regular expression syntax. For example, producing an ⟨*atom*⟩ from a concatentation RE requires wrapping its ⟨*regexp*⟩ form with (?:␣).

```
; emit-regexp : RE -> String
(define (emit-regexp re)
  (match re
    [(re:or res) (string-join (map emit-pces res) "|")]
    [_ (emit-pces re #f)]))
; emit-pces : RE [Boolean] -> String
(define (emit-pces re [rec? #t])
  (match re
    [(re:cat res) (string-join (map emit-pces res) "")]
    [_ (emit-pce re rec?)]))
; emit-pce : RE [Boolean] -> String
(define (emit-pce re [rec? #t])
  (match re
    [(re:repeat #f re) (format "~a*" (emit-atom re))]
    [(re:repeat #t re) (format "~a+" (emit-atom re))]
    [_ (emit-atom re rec?)]))
; emit-atom : RE [Boolean] -> String
(define (emit-atom re [rec? #t])
  (match re
    [(re:report re) (format "(~a)" (emit-regexp re))]
    [(re:chars ranges) (format "[~a]" (string-join (map emit-
range ranges) ""))]
    [_ (cond [rec? (format "(?:~a)" (emit-regexp re))]
             [else (error 'emit-regexp "bad RE: ~e" re)])]))
; emit-range : Range -> String
(define (emit-range r)
  (match r
    [(rng c c) (emit-char c)]
    [(rng lo hi) (format "~a-~a" (emit-char lo) (emit-
char hi))]))
; emit-char : Char -> String
(define (emit-char c)
  (define (special? c) (for/or ([sp (in-string "()*+?[]{}.^\\|")]) (eqv? c sp)))
  (if (special? c) (string #\\ c) (string c)))
```

Here is an example:

```
> (emit-regexp
   (re:repeat #f
    (re:cat (list (re:report (re:repeat #t (re:chars (list (rng #\a #\z)))))
                  (re:repeat #t (re:chars (list (rng #\space #\space))))))))
"(?:([a-z]+)[ ]+)*"
```

So, the ergonomics leave a bit to be desired. It would be possible to improve the interface by using friendlier functions instead of the raw AST constructors, of course. Or we could even define an S-expression notation and parse it into the RE type using match. All of potentially incurs additional run-time overhead, and there is also the overhead of pregexp call itself.

In any case, this code represents a complete, self-contained unit of functionality. Let's wrap up the code above as module:

```
(module re-ast racket/base
  (require racket/match racket/string)
  (provide (struct-out re:or)
           (struct-out re:cat)
           ....
           emit-regexp)
  ....)
```

We can leave a friendlier front end as a task for a separate module.

9.1.2 A Macro Front-End

Let's add a macro "front end" to the RE type and emit-regexp function. Specifically, the macro should support a friendlier notation that it parses into a compile-time RE value, translates to a string, and converts to a pregexp literal, all at compile time.

Here is a shape for representing (a subset of) regular expressions:

```
;; RE ::= (or RE ...+)
;;      | (cat RE ...)
;;      | (* RE)
;;      | (+ RE)
;;      | (report RE)
;;      | (chars CharRange ...+)
;; CharRange ::= Char | [Char Char]
```

I have called the shape RE, the same as the RE type. In fact, the interpretation of the RE term is a compile-time RE value. We can import the RE type and emit-regexp function into the *compile-time environment* as follows:

```
(require (for-syntax 're-ast))
```

The re syntax class, then, should have a single attribute whose value is an RE value.

Before we define the syntax class, we should decide how to recognize the literals in the shape definition (aka *grammar*) above. In §8.3 "Quasiquotation", I said there are two options: symbolic literals and reference literals. In this case, I want to use names that are already defined by Racket, but their interpretations here have nothing to do with their Racket meanings. More importantly, I don't plan to support macro-like extensions to this syntax, which is one major reason to recognize literals by reference instead of symbolically. So let's recognize the RE literals symbolically. We can do that by declaring them with `#:datum-literals` instead of `#:literals`. Here are the syntax class definitions:

```
(begin-for-syntax
  (define-syntax-class char-range
    #:attributes (ast) ; Range
    (pattern c:char
             #:attr ast (let ([c (syntax->datum #'c)]) (rng c c)))
    (pattern [lo:char hi:char]
             #:attr ast (rng (syntax->datum #'lo) (syntax-
>datum #'hi))))
  (define-syntax-class re
    #:attributes (ast) ; RE
    #:datum-literals (or cat * + report chars)
    (pattern (or e:re ...+)
             #:attr ast (re:or (datum (e.ast ...))))
    (pattern (cat e:re ...)
             #:attr ast (re:cat (datum (e.ast ...))))
    (pattern (* e:re)
             #:attr ast (re:repeat #f (datum e.ast)))
    (pattern (+ e:re)
             #:attr ast (re:repeat #t (datum e.ast)))
    (pattern (report e:re)
             #:attr ast (re:report (datum e.ast)))
    (pattern (chars r:char-range ...+)
             #:attr ast (re:chars (datum (r.ast ...))))))
```

The `my-px` macro simply calls `emit-regexp` on the parsed RE value, then calls `pregexp` to convert that to a (compile-time) regular expression value.

```
; (my-px RE) : Expr
(define-syntax my-px
  (syntax-parser
    [(_ e:re)
     #`(quote #,(pregexp (emit-regexp (datum e.ast))))]))
```

Note that we use `quote` to wrap the value returned by `pregexp`.

Here is the example from the previous section translated to use the macro's notation:

```
> (my-px (* (cat (report (+ (chars [#\a #\z])))
                 (+ (chars #\space)))))
#px"(?:([a-z]+)[ ]+)*"
```

Exercise 20: Update the RE shape with the following case:

```
;; RE ::= ... | String
```

You can use string as a syntax class annotation to recognizes string terms.

A string is interpreted as the concatenation of the singleton character sets of each character in the string. For example:

```
(my-px (* "ab"))
; expect #px"(?:ab)*"
```

9.2 The Static Reference Shape

The Static[T] shape is a parameterized shape that recognizes identifiers that refer to compile-time information of type T. The interpretation of a Static[T] reference is the compile-time T value.

The corresponding static syntax class is parameterized by a predicate and a description. The syntax class matches an identifier if the identifier is bound in the *normal environment* to a *compile-time* value accepted by the predicate; the value attribute contains the compile-time value.

That is, the Static shape contains identifiers bound with define-syntax, let-syntax, and so on. I'll call this a *static binding*, as opposed to a *variable bindings* created by define, let, and so on. Static bindings are also created by macros such as struct: the name of a struct type carries compile-time information about the struct type, including references to its predicate and accessor functions. This information is used by macros like match to implement pattern matching; it is also used by struct-out to get the names to export. (And remember, a *static* binding in the *normal environment* is different from a *variable* binding in the *compile-time environment*, even though both refer to compile-time values.)

Terminology: I don't like "bound as syntax". I'm not totally happy with "bound statically" either, though. "bound to compile-time information" is too verbose. "bound at compile time" is wrong. Alternatives?

Notation: ~ or :: or ??

Let's extend our little regular expression language with the ability to define and use RE names.

Here is the shape of the definition form:

```
;; (define-re x:Id RE) : Body[{x ~ RE}]
```

I'm using the notation from §7.3 "Shapes, Types, and Scopes (⋆)" to indicate the bindings introduced by a Body term, but I have extended it with the notation x ~ RE to mean that x is bound *statically* to a compile-time value of type RE — as opposed to x : T, which means that x is bound as a variable to a run-time value of type T.

The variants of the RE type are represented by prefab structs, which are readable and — more importantly — quoteable. So we can implement the definition macro as follows:

```
(define-syntax define-re
  (syntax-parser
    [(_ name:id e:re)
     #`(define-syntax name '#,(datum e.ast))]))
```

That is, `define-re` parses the RE term to get a compile-time RE value. But it cannot directly create a static binding for `name`; it must do so by expanding to a `define-syntax` term. So the macro must convert the compile-time RE value that it has now into an expression that will produce an equivalent value later, when the macro expander processes the `define-syntax` form. Since the value is `read`able, we can do that with `quote`. (If the value were not readable, then this would create "3D syntax", and modules using the macro would fail to compile. Or more precisely, compilation would succeed but the compiler would be unable to serialize the compiled code to a `.zo` file.)

To allow references to static RE bindings, we extend the RE shape as follows:

```
;; RE ::= ... | Static[RE]
```

The `re` syntax class needs a new pattern for references to RE names, and that pattern needs a helper predicate to recognize RE values. The existing patterns are unchanged.

```
(begin-for-syntax
  ; re-ast? : Any -> Boolean
  ; Shallow check for RE AST constructor. Sufficient?
  (define (re-ast? v)
    (or (re:or? v) (re:cat? v) (re:repeat? v) (re:report? v) (re:chars? v)))

  (define-syntax-class re
    #:attributes (ast) ; RE
    #:datum-literals (or cat * + report chars)
    ....
    (pattern (~var name (static re-ast? "name bound to RE"))
             #:attr ast (datum name.value))))
```

Now we can define intermediate RE names and compose them into more complicated regular expressions:

```
> (define-re word (+ (chars [#\a #\z])))
> (define-re spacing (+ (chars #\space)))
> (my-px (* (cat (report word) spacing)))
#px"(?:([a-z]+)[ ]+)*"
```

If we attempt to refer to a name that is not defined as a RE, then we get an appropriate error:

```
> (my-px (cat word list))
```

```
eval:28:0: my-px: expected name bound to RE
  at: list
  in: (my-px (cat word list))
  parsing context:
   while parsing re
    term: list
    location: eval:28:0
   while parsing re
    term: (cat word list)
    location: eval:28:0
```

We can inspect the compile-time value bound to an `RE` name by using `syntax-local-value`, which is is the low-level mechanism underneath `static`:

```
> (phase1-eval (syntax-local-value (quote-syntax word)))
'#s(re:repeat #t #s(re:chars (#s(rng #\a #\z))))
```

9.3 Static Information with Multiple Interfaces

There are still a few issues:

1. It would be nice if we could also use `RE` names like `word` and `spacing` as variables.
2. The shallow `re-ast?` test doesn't guarantee that the name was defined using `define-re`. After all, anyone can create a prefab struct named `re:repeat`.
3. This design does not allow forward references: an `RE` name must be defined before it is used. But it is often preferable to define complex objects in a top-down order.

We can fix the first two issues by adding a generative (that is, not prefab) struct wrapper around the `RE` value, and making it support the procedure interface so it acts as an identifier macro. The next section shows how to support forward references, at least in most contexts.

The macro expander considers any name that is statically bound to a procedure to be a macro. It invokes the macro's transformer on uses of the macro both in operator position and as a solitary identifier. A macro that allows being used as a solitary identifier is called an *identifier macro*. (If the macro's value is a `set!-transformer`, it is also invoked when the macro is used as the target of a `set!` expression.)

In Racket, any non-prefab struct can act as a procedure by implementing the *procedure interface*, represented by the `prop:procedure` struct type property. The macro system defines other interfaces, such as `prop:rename-transformer` and `prop:set!-transformer`, and macros can also define their own interfaces. For example, the `struct` form defines an interface, `prop:struct-info`, for representing compile-time information about struct types; the match macro defines an interface, prop:match-expander, for implementing new match pattern forms; and so on. Thus one name can carry multiple kinds of static information and behavior by being bound to a struct type that implements multiple interfaces.

We could even define and export our own interface for values representing RE names. But that would conflict with our goal of restricting RE names to those defined through our define-re macro, which enforces invariants on the values carried by RE names. Furthermore, it commits us to a representation; if we change how we represent information (as we will in the next section), it can break code that relies on the public interface. These problems can be mitigated with well-formedness checks and adapters, but that is additional effort and complexity, and it often doesn't completely fix the problem. In this example, the costs and risks don't seem worth the (absent) benefits. Another possibility is to define an interface but keep it private, only using it within a library. That avoids the problems above. It still doesn't seem useful in this particular example, though.

So, we will define a new struct type that implements the procedure interface so RE names can be used as expressions, but our macros will recognize the struct type specifically, without going through an additional interface indirection. Here is the definition:

```
(begin-for-syntax
  ; A RE-Binding is (re-binding RE (Syntax -> Syntax))
  (struct re-binding (ast transformer)
    #:property prop:procedure (struct-field-index transformer)))
```

When used as a procedure, an re-binding instance just forwards the call to its transformer field, so when define-re constructs an re-binding instance, it must provide a suitable transformer function. We can use the make-variable-like-transformer library function to construct an identifier macro that always produces the same expansion. Here is the updated define-re:

```
(require (for-syntax syntax/transformer))
(define-syntax define-re
  (syntax-parser
    [(_ name:id e:re)
     #`(define-syntax name
         (re-binding (quote #,(datum e.ast))
                     (make-variable-like-transformer
                      (quote-syntax (my-px name)))))]))
```

Now we must update re to extract the RE AST value in the RE-name case. Again, the other variants remain unchanged:

```
(begin-for-syntax
  (define-syntax-class re
    #:attributes (ast) ; RE
    #:datum-literals (or cat * + report chars)
```

```
....
(pattern (~var name (static re-binding? "name bound to RE"))
         #:attr ast (re-binding-ast (datum name.value)))))
```

Now the following example works; we can use `word` and `spacing` like variables:

```
> (define-re word (+ (chars [#\a #\z])))
> (define-re spacing (+ (chars #\space)))
> (define-re word+spacing (cat (report word) spacing))
> (list word spacing (my-px (* word+spacing)))
'(#px"[a-z]+" #px"[ ]+" #px"(?:([a-z]+)[ ]+)*")
```

We can also verify that the compile-time information stored by an `RE` name is no longer a prefab struct; it is an opaque wrapper which prints as a procedure:

```
> (phase1-eval (syntax-local-value (quote-syntax word)))
#<procedure:...tax/transformer.rkt:19:3>
```

Exercise 21: Update the definition of `re-binding` so that instances of the struct print as `"#<RE>"`. You can do this by implementing the `prop:custom-write` interface.

```
(phase1-eval (format "~s" (syntax-local-eval (quote-
syntax word))))
; expect "#<RE>"
```

Exercise 22 (⋆): Update the definition of `re-binding` so that it also acts as a match pattern name. As a match pattern, it takes some number of pattern arguments; these are the patterns used to match the regular expression's `report` results. That is, an `RE` name used as a match pattern expands like this:

```
(word+spacing pat)
⇒
(pregexp word+spacing (list _ pat))
```

See `match` for an explanation of the syntax of match patterns; see `prop:match-expander` for an explanation of the interface.

Exercise 23 (⋆⋆): Update your solution to Exercise 22 to check that when used as a match pattern, the `RE` name receives the correct number of arguments. That is, the first example below should succeed (because `word+spacing` has one `report`), but the others should cause an error:

```
(word+spacing the-word)         ; okay
(word+spacing the-word extra)   ; error: wrong number of patterns,
expected 1
(spacing the-spaces)            ; error: wrong number of patterns,
expected 0
```

9.4 Two-Pass Expansion

To support forward references requires knowing a little about how Racket processes definitions.

The Racket macro expander processes definition contexts (module bodies, `lambda` bodies, and so on) in two passes. The first pass discovers definitions; the second pass expands (remaining) expressions.

In the first pass, Racket expands each body term until it reaches a core form, but it does not recur into the core form's sub-expressions. Once it reaches a core form, it does a shallow case analysis. If the form is `define-values`, it marks the names as variables in the local environment. If the form is `define-syntaxes`, it evaluates the right-hand side as a compile-time expression and binds the names statically. If the form is `begin`, it flattens it away and recurs on the contents. (So a macro can expand into multiple definitions by grouping them with `begin`.) Otherwise the form is an expression form, and it leaves that until the next pass. After the case analysis, it continues to the next body term.

In the second pass, the expander knows all of the names bound in the recursive definition context, both variable names and static names. The expander then processes the remaining expressions: the sub-expressions of core expression forms and the right-hand sides of `define-values` definitions.

9.4.1 Forward References

How can we support forward references for compile-time data?

Since static definitions are evaluated in order, the static information for a `RE` name cannot just contain an `RE` AST value. It must contain a thunk or promise or something similar that allows us to get the AST value on demand. Let's use a promise. Here is the updated struct definition:

```
(require (for-syntax racket/promise))
(begin-for-syntax
  ; A RE-Binding is (re-binding (Promise RE) (Syntax -> Syntax))
  (struct re-binding (astp transformer)
    #:property prop:procedure (struct-field-index transformer)))
```

Now `define-re` cannot eagerly parse the `RE` term; instead, it must create a promise that parses it later. Here's one implementation:

```
(begin-for-syntax
  ; parse-re-from-def : Syntax -> RE
  ; Receives the entire `define-re` term.
  (define (parse-re-from-def stx)
```

```
    (syntax-parse stx
      [(_ _ e:re) (datum e.ast)])))

(define-syntax define-re
  (syntax-parser
    [(_ name:id _)
     #`(begin
         (define-syntax name
           (re-binding (delay (parse-re-from-def (quote-
syntax #,this-syntax)))
                       (make-variable-like-transformer
                        (quote-syntax (my-px name)))))
         (void (my-px name)))]))
```

Within a `syntax-parser` clause, `this-syntax` is bound to the syntax object currently being parsed. In this case, that is the syntax of the `define-re` use. The reason for passing the whole definition syntax to `parse-re-from-def` instead of just the `RE` term is that by default `syntax-parse` reports syntax errors using the leading identifier of its argument as the "complaining party". (This behavior can be overridden with the `#:context` argument, though.)

Why does the expansion include `(void (my-px name))`? That expression ensures that the promise eventually gets forced. Since it occurs within an expression, it is delayed until pass two, when all forward references should have been defined. If we left it out, then a syntactically invalid `RE` definition would be accepted as long as it was never used.

Finally, we must update `re` to force the promise from a `RE` name. Here is a basic implementation:

```
(begin-for-syntax
  (define-syntax-class re
    #:attributes (ast) ; RE
    #:datum-literals (or cat * + report chars)
    ....
    (pattern (~var name (static re-binding? "name bound to RE"))
             #:attr ast (force (re-binding-
astp (datum name.value))))))
```

One flaw in this implementation is that if it is given a recursive `RE` definition, it produces an internal error about re-entrant promises. Here is a version that uses a parameter to detect that situation and signals a better error:

```
(begin-for-syntax
  ; running : (Parameter (Promise RE))
  ; Currently running RE promises, used to detect cycles.
  (define running (make-parameter null))
```

```
(define-syntax-class re
  #:attributes (ast) ; RE
  #:datum-literals (or cat * + report chars)
  ....
  (pattern (~var name (static re-binding? "name bound to RE"))
           #:attr ast (let ([p (re-binding-
ast (datum name.value))])
                        (cond [(member p (running)) #f]
                              [else (parameterize ((running (cons p (running))))
                                      (force p))]))
           #:fail-when (if (datum ast) #'name #f) "recursive
RE")))
```

With those changes, forward references work, at least within modules and within internal definition contexts like `let` bodies:

```
> (let ()
    (define-re para (* (cat word spacing)))
    (define-re word (+ (chars [#\a #\z])))
    (printf "word = ~s\n" word)
    (define-re spacing (+ (chars #\space)))
    para)
word = #px"[a-z]+"
#px"(?:[a-z]+[ ]+)*"
```

We get reasonable messages for the following error cases:

```
> (let ()
    (define-re uses-undef (cat (chars #\a) undef))
    'whatever)
eval:52:0: define-re: expected name bound to RE
  at: undef
  in: (define-re uses-undef (cat (chars #\a) undef))
  parsing context:
   while parsing re
    term: undef
    location: eval:52:0
   while parsing re
    term: (cat (chars #\a) undef)
    location: eval:52:0
> (let ()
    (define-re rec (cat (chars #\a) rec))
    'whatever)
eval:53:0: define-re: recursive RE
```

```
at: rec
in: (define-re rec (cat (chars #\a) rec))
parsing context:
  while parsing re
    term: rec
    location: eval:53:0
  while parsing re
    term: (cat (chars #\a) rec)
    location: eval:53:0
```

There are other ways we could manage the delayed resolution of RE names. For example, we could extend the AST type with a new variant for names, eagerly parse most of the AST and create promises only for instances of the name variant. One benefit of that approach is that most RE syntax errors could be caught when the definition is processed instead of when the promise is forced. Some drawbacks are that it involves either changing the RE type or creating a substantially similar RE-With-Promises type, and it requires adding a new function to traverse the AST forcing the name nodes.

9.4.2 The Peculiarities of Scoping in Two-Pass Expansion

The two-pass expansion and its treatment of macros means that definition contexts in Racket are not purely recursive; they also have a slight sequential aspect. Consider the behavior of the following example:

```
> (let ()
    (define-syntax m
      (syntax-parser
        [(_ e:expr) #'(printf "outer ~s\n" e)]))
    (let ()
      (m (begin (m 123) 456))
      (define-syntax m
        (syntax-parser
          [(_ e:expr) #'(printf "inner ~s\n" e)]))
      (m 789)))
inner 123
outer 456
inner 789
```

Simple recursive scoping would predict that all three references to `m` in the inner `let` body refer to the inner definition of `m`. But the first use of `m` is head-expanded before the inner definition of `m` is discovered, so it refers to the outer definition. Its argument, though, is not expanded until pass two, so it refers to the inner `m`, as does the third use of `m`. The third use of `m` is expanded before the second, though.

What if we simply delete the outer macro definition?

```
> (let ()
    (let ()
      (m (begin (m 123) 456))
      (define-syntax m
        (syntax-parser
          [(_ e:expr) #'(printf "inner ~s\n" e)]))
      (m 789)))
inner 123
inner 456
inner 789
```

Now in pass one the expander assumes that the first use of `m` is a function application, and `m` is a variable that might be defined later. So it saves the whole expression for pass two. Then in pass two it realizes that the expression is not a function application but a macro application, and it expands the macro. This is good, right? It is what one would expect given a macro definition in a recursive scope.

But there are limits. Consider the following example, where the macro produces a definition instead of an expression:

```
> (let ()
    (m a)
    (define-syntax m
      (syntax-parser
        [(_ x:id) #'(define x 1)]))
    (m b)
    (+ a b))
eval:56:0: define: not allowed in an expression context
  in: (define a 1)
```

As in the previous example, the macro expansion initially classifies the first use of `m` as a function application. In the second pass, though, when it expands the macro, it expands it in a strict expression context. That is because it is unwilling to make further changes to the environment in the second pass; it is frozen at the end of the first pass.

The greatest scoping peculiarities of definition contexts arise from macro names that are shadowed in the middle of an inner scope.

Lesson: *Don't shadow macro names.*

Unfortunately, many names in Racket that seem like variable names are actually implemented as macro bindings. One example is functions with keyword arguments, to reduce run-time overhead for keyword checking and default arguments. Another example is bindings exported with `contract-out`, to compute the negative blame party from the use site.

Lesson: *As much as possible, avoid shadowing entirely.*

10 Unhygienic Macros

Recall the definition of a hygienic macro: definition-site binders do not capture use-site references, and use-site binders do not capture definition-site references. Hygienic macros can still implement binding forms (recall `my-and-let`, for example, from §3.4 "The Identifier Shape"), but the bound names must be given as arguments.

Sometimes, though, it is useful for a macro to bind names that are visible to the macro use site without receiving the names as explicit arguments. Such macros are *unhygienic*; we also say that they "break hygiene". Unhygienic macros are mainly divided into two groups; I'm going to call them clean unhygienic macros and unclean unhygienic macros, and you can't stop me.

10.1 Clean Unhygienic Macros

A *clean unhygienic macro* defines names that are not given as `Id` arguments, but are based on one or more `Id` arguments.

A good example of a clean unhygienic macro is `struct`: it defines the predicate and accessor functions (as well as a few other names) based on the identifier given as the struct name and the identifers naming the fields. A greatly simplified version of `struct` could be given the following shape:

```
;; (struct s:Id (f:Id ...)) : Body[{s,s?,s-f...}]
```

As an example, let's design a macro `my-hash-view`, which puts a `struct`-like interface on symbol-keyed hashes. It has the following shape:

```
;; (my-hash-view v:Id (f:Id ...)) : Body[{v,v?,v-f...}]
```

It should have the following behavior:

```
(my-hash-view point (x y))
; defines point, point?, point-x, point-y
(point 1 2)
; expect (hash 'x 1 'y 2)
(point? (hash 'x 3 'y 4))
; expect #t
(point? (hash 'x 3 'y 4 'z 5))
; expect #t
(point? (hash 'x 6))
; expect #f
(point-x (hash 'x 7 'y 8))
; expect 7
```

Let's consider what code we could use to implement the intended behavior.

```
(begin
  (define (point x y)
    (hash 'x x 'y y))
  (define (point? v)
    (and (hash? v) (hash-has-key? v 'x) (hash-has-key? v 'y)))
  (define (point-x v)
    (unless (point? v)
      (raise-argument-error 'point-x "point?" v))
    (hash-ref v 'x))
  (define (point-y v)
    (unless (point? v)
      (raise-argument-error 'point-y "point?" v))
    (hash-ref v 'y)))
```

We need to produce the identifiers `point?`, `point-x`, and `point-y`. This code also has the string literal `"point?"`; we could compute it at run time (as we did in §1.2 "Designing Your First Macro"), but in this example let's go ahead and compute it at compile time. The other part of the code that is a bit tricky to produce is the body of the constructor function: `(hash 'x x 'y y)`. The `hash` arguments do not consist of a single repeated term, but rather each repetition consists of two terms. Fortunately, Racket's syntax templates support multi-term repetition using the `~@` template form.

Before we continue to the implementation of the macro, we can also use this hand-expansion to run our tests, to check that the expansion works before we automate its generation with the macro.

```
> (check-equal? (point 1 2) (hash 'x 1 'y 2))
> (check-pred point? (hash 'x 3 'y 4))
> (check-pred point? (hash 'x 3 'y 4 'z 5))
> (check-equal? (point? (hash 'x 6)) #f)
> (check-equal? (point-x (hash 'x 7 'y 8)) 7)
> (check-exn #rx"point-x: contract violation"
             (lambda () (point-x (hash 'z 9))))
```

The tests pass, so let's move on the the macro.

Given the identifier representing the use-site name `point`, how do we compute an identifier `point?` that acts like it also came from the macro use site? Using ordinary Racket functions we can compute the symbol `'point?` given the symbol `'point`. The extra step the macro must perform is to transfer the *lexical context* from the original `point` identifier to the new identifier. The primitive mechanism for doing that is `datum->syntax`: its first argument is an existing syntax object to take the lexical context from, and the second argument is a datum to wrap as the new syntax object. So the following is the process for computing the `point?` identifier from the `point` identifier:

```
(define point-id #'point)
(define point-symbol (syntax->datum point-id))
(define point?-symbol (string->symbol (format "~a?" point-
symbol)))
(define point?-id (datum->syntax point-id point?-symbol))
```

The format-id automates this process. It takes the lexical context source object first, then a restricted format string (allowing only ~a placeholders), and then the format strings arguments. Unlike format, format-id automatically unwraps identifiers in the format string arguments to their symbol contents.

```
(define point?-id (format-id point-id "~a?" point-id))
```

Additionally, format-id with the #:subs? #t option builds the identifier with a syntax property (a way of attaching extra information to a syntax object) indicating the positions of the original identifier components. This information lets, for example, DrRacket draw binding arrows to parts of identifiers.

```
(define point?-id (format-id point-id "~a?" point-id #:subs? #t))
```

Finally, instead of using quasisyntax and unsyntax (#` and #,) to insert the results of compile-time computation into syntax templates, we can use #:with or with-syntax to bind secondary syntax pattern variables to the computed terms.

Here is the macro definition:

```
(define-syntax my-hash-view
  (syntax-parser
    [(_ name:id (field:id ...))
     #:with name? (format-id #'name "~a?" #'name #:subs? #t)
     #:with name?-string (format "~a?" (syntax-
>datum #'name)) ; implicit datum->syntax
     #:with (name-field ...) (for/list ([fieldname (in-
list (datum (field ...)))])
                               (format-id #'name "~a-
~a" #'name fieldname #:subs? #t))
     ; name? : Id, name?-string : Datum, (name-field ...) : (Id
...)
     #'(begin
         (define (name field ...)
           (hash (~@ (quote field) field) ...))
         (define (name? v)
           (and (hash? v) (hash-has-key? v (quote field)) ...))
         (define (name-field v)
           (unless (name? v)
```

```
              (raise-argument-error (quote name-
field) (quote name?-string) v))
             (hash-ref v (quote field)))
           ...)]))
```

Let's run the tests against the macro implementation:

```
; (my-hash-view point (x y)))
> (check-equal? (point 1 2) (hash 'x 1 'y 2))
> (check-pred point? (hash 'x 3 'y 4))
> (check-pred point? (hash 'x 3 'y 4 'z 5))
> (check-equal? (point? (hash 'x 6)) #f)
> (check-equal? (point-x (hash 'x 7 'y 8)) 7)
> (check-exn #rx"point-x: contract violation"
             (lambda () (point-x (hash 'z 9))))
```

Exercise 24: The `#:with name?-string` binding in the definition above implicitly converts the string result of `format` into a syntax object. That's okay, as long as we treat `name?-string` as a `Datum`. What happens if we treat it like an `Expr` instead? Find out by replacing `(quote name?-string)` with `name?-string` in the macro's syntax template.

Exercise 25: Update the implementation of `my-hash-view` to allow field names to have different hash keys. That is, generalize the shape to the following:

```
;; (my-hash-view v:Id [fs:FieldSpec ...]) : Body[{v,v?,v-
fs.fn...}]
;; where FieldSpec ::= fn:Id | [fn:Id #:key Datum]
```

Here is an example to illustrate the intended behavior:

```
(my-hash-view post (author [link #:key resource_href]))
(define post1 (hash 'author "Ryan" 'resource_href "/malr/unhygienic.html"))
(post-link post1) ; expect "/malr/unhygienic.html"
```

Hint: use the common meaning interface strategy.

Exercise 26: Update the implementation of `my-hash-view` so that the hash view name acts both as a constructor and as a `match` pattern name. That is, the hash view name should be statically bound to a compile-time struct implementing both the procedure interface and the `match` expander interface. You should define the actual constructor function with a different name and expand to it using `make-variable-like-transformer`. For the match expander, use the `?` and `app` match pattern forms. That is, as a match pattern, `point` behaves as follows:

```
(point x-pat y-pat)
⇒
(? point? (app point-x x-pat) (app point-y y-pat))
```

Exercise 27 (⋆): Update your solution to Exercise 26 to also support hash view extension (or "subtyping"). That is, the value statically bound to hash-view name must support three interfaces: the procedure interface, the `match` expander interface, and a private interface that carries enough information to support view extension.

Here are some examples to illustrate the expected behavior:

```
(my-hash-view point (x y))
(my-hash-view point3 #:super point (z))
(define p3 (point3 1 2 3))
(point? p3) ; expect #t
(point3? p3) ; expect #t
(point-x p3) ; expect 1
(point3-z p3) ; expect 3
(match p3 [(point x y) (+ x y)]) ; expect 3
(match p3 [(point3 x y z) (+ x y z)]) ; expect 6
```

10.2 Unclean Unhygienic Macros

An *unclean unhygienic macro* defines names that are not based on any `Id` arguments.

The canonical example of an unclean unhygienic macro is a `while` loop that binds the name `break` to an escape continuation to exit the loop.

What lexical context should the macro use to create the `break` binder? The best candidate here is the lexical context of the whole macro use. In a `syntax-parser` form, this is available through the name `this-syntax`. (You might wonder whether `this-syntax` is bound unhygienically. It isn't. In fact, we'll talk about the mechanism it uses in §10.4 "Syntax Parameters".)

Here is the macro definition:

```
; (while Expr Expr{break} ...+) : Expr
(define-syntax while
  (syntax-parser
    [(_ condition:expr loop-body:expr ...+)
     #:with break (datum->syntax this-syntax 'break)
     #'(let/ec break
         (let loop ()
           (when condition
             loop-body ...
             (loop))))]))
```

With this macro, we can finally write FORTRAN in Racket:

```
> (define ns '(2 3 4 5 6))
```

```
> (define sum 0)
> (while (pair? ns)
    (when (integer? (sqrt (car ns))) (break))
    (set! sum (+ sum (car ns)))
    (set! ns (cdr ns)))
> sum
5
```

Now let's write the macro `forever` that uses `while` as the helper macro. That is:

```
(forever loop-body)
⇒
(while #t loop-body)
```

It should be trivial, right? Here's a definition:

```
; (forever Expr{break} ...+) : Expr
(define-syntax forever
  (syntax-parser
    [(_ loop-body:expr ...+)
     #'(while #t loop-body ...)]))
```

But if we try to use `break` in the loop body, this happens:

```
> (define counter 0)
> (forever
    (set! counter (add1 counter))
    (unless (< counter 5) (break))
    (printf "counter = ~s\n" counter))
counter = 1
counter = 2
counter = 3
counter = 4
break: undefined;
 cannot reference an identifier before its definition
  in module: top-level
```

In a module, this wouldn't even compile, because `break` is unbound.

What went wrong? Here is one explanation: The `forever` example expands into a use of `while`, which expands into code that binds `break` with the lexical context of the `while` expression. But the lexical context of the `while` expression is from the definition site of `forever`, not the use site of `forever` in the example! Given that those are not necessarily the same, there's no reason to expect the example to work.

On the other hand, it's not clear what makes the two sites different, either. What is a "site", anyway? The definition of forever and the example use of forever are both top level interactions (of this Scribble document's evaluator, specifically); what makes them distinct?

We need to refine our definition of hygiene slightly. Each time a macro is invoked, it is considered to have a different "site". More precisely, the meaning of *references* in the macro's syntax template is determined by its definition site, but an extra marker is added that distinguishes binders introduced by different macro invocations. In the terminology of Racket's hygiene model, this extra marker is called a macro-introduction scope.

We can "fix" the implementation of forever by adjusting the lexical context on the syntax object representing the use of the while macro (but not on any of its subterms) to be the same as the use of the forever macro. We do that by using syntax-e to unwrap just the outer layer of syntax, and then we use datum->syntax to rebuild it with the lexical context of this-syntax. Here is the implementation:

```
; (forever Expr{break} ...+) : Expr
(define-syntax forever
  (syntax-parser
    [(_ loop-body:expr ...+)
     (define code
       #'(while #t loop-body ...))
     (datum->syntax this-syntax (syntax-e code))]))
```

Now the example works:

```
> (define counter 0)
> (forever
    (set! counter (add1 counter))
    (unless (< counter 5) (break))
    (printf "counter = ~s\n" counter))
counter = 1
counter = 2
counter = 3
counter = 4
```

With this approach, break is visible to the loop body (well, assuming that the loop body terms have the same lexical context as the term representing the whole call to forever, which is not necessarily true), but it is not visible to the code introduced by the forever macro.

Here's another approach that works if we want to use break in the macro as well as making it visible to the loop body:

```
; (do-while Expr Body{break} ...+) : Expr
(define-syntax do-while
```

```
(syntax-parser
  [(_ condition:expr loop-body:expr ...+)
   #:with break/user (datum->syntax this-syntax 'break)
   #'(while #t
       (let ([break/user break])
         loop-body ...)
       (unless condition (break)))]))
```

Lesson: *Unhygienic macros are difficult to use as helper macros — that is, as the targets of expansion.*

10.3 Optionally-Hygienic Macros

Consider Racket's `require` form. For example,

```
(require racket/list)
```

locally binds the names `first`, `second`, and so on, even though those names are not given as binder `Id` arguments to `require`. In fact, `require` is acting as an unclean unhygienic binding form here — its argument, `racket/list` is an identifier, but `require`'s argument shape is `RequireSpec`, which has a `ModulePath` variant, which has an identifier variant. We could also consider `(require (lib "racket/list.rkt"))`, which means the same thing.

On the other hand, in the following,

```
(require (only-in racket/list first [last final]))
```

the `first` identifier is used for the binding of the `first` import, and the `final` identifier is used for the binding of the import that `racket/list` exports as `last`. So this particular usage of `require` is hygienic!

One way to mitigate the difficulty that unhygienic macros cause is to give them hygienic options. For example, we could extend `while` with an optional clause for specifying the name to bind to the escape continuation. If the clause is present, the macro binds the given name, and it is hygienic; if the clause is absent, it generates the name unhygienicially. Here is the optional clause shape:

```
;; MaybeBreakClause ::= ε | #:break Id
```

Instead of defining a (splicing) syntax class for it, though, let's just handle it inline within the macro's syntax pattern using the `~optional` pattern form. If an `~optional` pattern is absent, then all of its pattern variables are bound to the value `#f` (note: not the syntax object representing the term `#f`). Normally, only syntax-valued attributes can be used within syntax templates, but the template form ~? can dynamically "catch" false-valued attributes in its first sub-template and fall back to its second sub-template. We can define the macro as follows:

```
; (while Expr MaybeBreakClause Expr{break} ...+) : Expr
(define-syntax while
  (syntax-parser
    [(_ condition:expr (~optional (~seq #:break break-name:id))
        loop-body:expr ...+)
     #:with default-break (datum->syntax this-syntax 'break)
     #'(let/ec (~? break-name default-break)
         (let loop ()
           (when condition
             loop-body ...
             (loop))))]))
```

Here is an example:

```
> (define n 2022)
> (while #t #:break stop
    (cond [(= n 1) (printf "\n") (stop)]
          [(even? n) (printf "↑") (set! n (quotient n 2))]
          [(odd? n) (printf "↓") (set! n (add1 (* n 3)))]))
↑↓↑↓↑↑↑↓↑↑↓↑↓↑↑↓↑↑↓↑↑↓↑↓↑↓↑↓↑↑↑↑↑↓↑↓↑↑↓↑↑↓↑↑↑↑↓↑↓↑↓↑↑↓↑↑↑↓↑↑↑↑
```

Here is the equivalent definition with a separate syntax class:

```
(begin-for-syntax
  (define-splicing-syntax-class maybe-break-clause
    #:attributes (break-name) ; (U #f Syntax[Id])
    (pattern (~seq #:break break-name:id))
    (pattern (~seq) #:attr break-name #f)))

; (while Expr MaybeBreakClause Expr{break} ...+) : Expr
(define-syntax while
  (syntax-parser
    [(_ condition:expr bc:maybe-break-clause
        loop-body:expr ...+)
     #:with default-break (datum->syntax this-syntax 'break)
     #'(let/ec (~? bc.break-name default-break)
         (let loop ()
           (when condition
             loop-body ...
             (loop))))]))
```

10.4 Syntax Parameters

Another alternative to unclean unhygienic macros is to define a single name that takes on different meanings in different contexts. This is analogous to run-time parameter values, so the feature is called a *syntax parameter*.