7 Contracts

This chapter provides a gentle introduction to Racket's contract system.

§7 "Contracts" in ***The Racket Reference*** provides more on contracts.

7.1 Contracts and Boundaries

Like a contract between two business partners, a software contract is an agreement between two parties. The agreement specifies obligations and guarantees for each "product" (or value) that is handed from one party to the other.

A contract thus establishes a boundary between the two parties. Whenever a value crosses this boundary, the contract monitoring system performs contract checks, making sure the partners abide by the established contract.

In this spirit, Racket encourages contracts mainly at module boundaries. Specifically, programmers may attach contracts to `provide` clauses and thus impose constraints and promises on the use of exported values. For example, the export specification

```
#lang racket

(provide/contract
  [amount positive?])
(define amount ...)
```

promises to all clients of the above module that the value of `amount` will always be a positive number. The contract system monitors the module's obligation carefully. Every time a client refers to `amount`, the monitor checks that the value of `amount` is indeed a positive number.

The contracts library is built into the Racket language, but if you wish to use `racket/base`, you can explicitly require the contracts library like this:

```
#lang racket/base
(require racket/contract) ; now we can write contracts

(provide/contract
  [amount positive?])
(define amount ...)
```

7.1.1 Contract Violations

If we bind `amount` to a number that is not positive,

```
#lang racket

(provide/contract
  [amount positive?])
(define amount 0)
```

then, when the module is required, the monitoring system signals a violation of the contract and blames the module for breaking its promises.

An even bigger mistake would be to bind `amount` to a non-number value:

```
#lang racket

(provide/contract
  [amount positive?])
(define amount 'amount)
```

In this case, the monitoring system will apply `positive?` to a symbol, but `positive?` reports an error, because its domain is only numbers. To make the contract capture our intentions for all Racket values, we can ensure that the value is both a number and is positive, combining the two contracts with `and/c`:

```
(provide/contract
  [amount (and/c number? positive?)])
```

7.1.2 Experimenting with Contracts and Modules

All of the contracts and modules in this chapter (excluding those just following) are written using the standard `#lang` syntax for describing modules. Since modules serve as the boundary between parties in a contract, examples involve multiple modules.

To experiment with multiple modules within a single module or within DrRacket's definitions area, use the `racket/load` language. The contents of such a module can be other modules (and `require` statements), using the longhand parenthesized syntax for a module (see §6.2.1 "The `module` Form"). For example, try the example earlier in this section as follows:

```
#lang racket/load

(module m racket
  (provide/contract [amount (and/c number? positive?)])
  (define amount 150))

(module n racket
```

```
  (require 'm)
  (+ amount 10))

(require 'n)
```

Each of the modules and their contracts are wrapped in parentheses with the `module` keyword at the front. The first form after `module` is the name of the module to be used in a subsequent `require` statement (where each reference through a `require` prefixes the name with a quote). The second form after `module` is the language, and the remaining forms are the body of the module. After all of the modules, a `require` starts one of the modules plus anything that is `requires`.

7.2 Simple Contracts on Functions

A mathematical function has a *domain* and a *range*. The domain indicates the kind of values that the function can accept as arguments, and the range indicates the kind of values that it produces. The conventional notation for a describing a function with its domain and range is

```
f : A -> B
```

where `A` is the domain of the function and `B` is the range.

Functions in a programming language have domains and ranges, too, and a contract can ensure that a function receives only values in its range and produces only values in its domain. A `->` creates such a contract for a function. The forms after a `->` specify contracts for the domains and finally a contract for the range.

Here is a module that might represent a bank account:

```
#lang racket

(provide/contract
  [deposit (-> number? any)]
  [balance (-> number?)])

(define amount 0)
(define (deposit a) (set! amount (+ amount a)))
(define (balance) amount)
```

The module exports two functions:

- `deposit`, which accepts a number and returns some value that is not specified in the contract, and

- `balance`, which returns a number indicating the current balance of the account.

When a module exports a function, it establishes two channels of communication between itself as a "server" and the "client" module that imports the function. If the client module calls the function, it sends a value into the server module. Conversely, if such a function call ends and the function returns a value, the server module sends a value back to the client module. This client–server distinction is important, because when something goes wrong, one or the other of the parties is to blame.

If a client module were to apply `deposit` to `'millions`, it would violate the contract. The contract-monitoring system would catch this violation and blame client for breaking the contract with the above module. In contrast, if the `balance` function were to return `'broke`, the contract-monitoring system would blame the server module.

A `->` by itself is not a contract; it is a *contract combinator*, which combines other contracts to form a contract.

7.2.1 Styles of `->`

If you are used to mathematical function, you may prefer a contract arrow to appear between the domain and the range of a function, not at the beginning. If you have read *How to Design Programs*, you have seen this many times. Indeed, you may have seen contracts such as these in other people's code:

```
(provide/contract
  [deposit (number? . -> . any)])
```

If a Racket S-expression contains two dots with a symbol in the middle, the reader rearranges the S-expression and place the symbol at the front, as described in §2.4.3 "Lists and Racket Syntax". Thus,

```
(number? . -> . any)
```

is just another way of writing

```
(-> number? any)
```

7.2.2 `any` and `any/c`

The `any` contract used for `deposit` matches any kind of result, and it can only be used in the range position of a function contract. Instead of `any` above, we could use the more specific contract `void?`, which says that the function will always return the `(void)` value. The

`void?` contract, however, would require the contract monitoring system to check the return value every time the function is called, even though the "client" module can't do much with the value. In contrast, `any` tells the monitoring system *not* to check the return value, it tells a potential client that the "server" module *makes no promises at all* about the function's return value, even whether it is a single value or multiple values.

The `any/c` contract is similar to `any`, in that it makes no demands on a value. Unlike `any`, `any/c` indicates a single value, and it is suitable for use as an argument contract. Using `any/c` as a range contract imposes a check that the function produces a single value. That is,

```
(-> integer? any)
```

describes a function that accepts and integer and returns any number of values, while

```
(-> integer? any/c)
```

describes a function that accepts an integer and produces a single result (but does not say anything more about the result). The function

```
(define (f x) (values (+ x 1) (- x 1)))
```

matches `(-> integer? any)`, but not `(-> integer? any/c)`.

Use `any/c` as a result contract when it is particularly important to promise a single result from a function. Use `any` when you want to promise as little as possible (and incur as little checking as possible) for a function's result.

7.2.3 Rolling Your Own Contracts

The `deposit` function adds the given number to the value of `amount`. While the function's contract prevents clients from applying it to non-numbers, the contract still allows them to apply the function to complex numbers, negative numbers, or inexact numbers, none of which sensibly represent amounts of money.

The contract system allows programmers to define their own contracts as functions:

```
#lang racket

(define (amount? a)
  (and (number? a) (integer? a) (exact? a) (>= a 0)))

(provide/contract
  ; an amount is a natural number of cents
```

```
  ; is the given number an amount?
  [deposit (-> amount? any)]
  [amount? (-> any/c boolean?)]
  [balance (-> amount?)])

(define amount 0)
(define (deposit a) (set! amount (+ amount a)))
(define (balance) amount)
```

This module defines an `amount?` function and uses it as a contract within `->` contracts. When a client calls the `deposit` function as exported with the contract `(-> amount? any)`, it must supply an exact, nonnegative integer, otherwise the `amount?` function applied to the argument will return `#f`, which will cause the contract-monitoring system to blame the client. Similarly, the server module must provide an exact, nonnegative integer as the result of `balance` to remain blameless.

Of course, it makes no sense to restrict a channel of communication to values that the client doesn't understand. Therefore the module also exports the `amount?` predicate itself, with a contract saying that it accepts an arbitrary value and returns a boolean.

In this case, we could also have used `natural-number/c` in place of `amount?`, since it implies exactly the same check:

```
(provide/contract
  [deposit (-> natural-number/c any)]
  [balance (-> natural-number/c)])
```

Every function that accepts one argument can be treated as a predicate and thus used as a contract. For combining existing checks into a new one, however, contract combinators such as `and/c` and `or/c` are often useful. For example, here is yet another way to write the contracts above:

```
(define amount/c
  (and/c number? integer? exact? (or/c positive? zero?)))

(provide/contract
  [deposit (-> amount/c any)]
  [balance (-> amount/c)])
```

Other values also serve double duty as contracts. For example, if a function accepts a number or `#f`, `(or/c number? #f)` suffices. Similarly, the `amount/c` contract could have been written with a `0` in place of `zero?`. If you use a regular expression as a contract, the contract accepts strings and byte strings that match the regular expression.

Naturally, you can mix your own contract-implementing functions with combinators like `and/c`. Here is a module for creating strings from banking records:

```
#lang racket

(define (has-decimal? str)
  (define L (string-length str))
  (and (>= L 3)
       (char=? #\. (string-ref str (- L 3)))))

(provide/contract
  ; convert a random number to a string
  [format-number (-> number? string?)]

  ; convert an amount into a string with a decimal
  ; point, as in an amount of US currency
  [format-nat (-> natural-number/c
                  (and/c string? has-decimal?))])
```

The contract of the exported function `format-number` specifies that the function consumes a number and produces a string. The contract of the exported function `format-nat` is more interesting than the one of `format-number`. It consumes only natural numbers. Its range contract promises a string that has a `.` in the third position from the right.

If we want to strengthen the promise of the range contract for `format-nat` so that it admits only strings with digits and a single dot, we could write it like this:

```
#lang racket

(define (digit-char? x)
  (member x '(#\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9 #\0)))

(define (has-decimal? str)
  (define L (string-length str))
  (and (>= L 3)
       (char=? #\. (string-ref str (- L 3)))))

(define (is-decimal-string? str)
  (define L (string-length str))
  (and (has-decimal? str)
       (andmap digit-char?
               (string->list (substring str 0 (- L 3))))
       (andmap digit-char?
               (string->list (substring str (- L 2) L)))))

....

(provide/contract
  ....
```

```
  ; convert an amount (natural number) of cents
  ; into a dollar based string
  [format-nat (-> natural-number/c
                  (and/c string?
                         is-decimal-string?))])
```

Alternately, in this case, we could use a regular expression as a contract:

```
#lang racket

(provide/contract
  ....
  ; convert an amount (natural number) of cents
  ; into a dollar based string
  [format-nat (-> natural-number/c
                  (and/c string? #rx"[0-9]*\\.[0-9][0-9]"))])
```

7.2.4 Contracts on Higher-order Functions

Function contracts are not just restricted to having simple predicates on their domains or ranges. Any of the contract combinators discussed here, including function contracts themselves, can be used as contracts on the arguments and results of a function.

For example,

```
(-> integer? (-> integer? integer?))
```

is a contract that describes a curried function. It matches functions that accept one argument and then return another function accepting a second argument before finally returning an integer. If a server exports a function `make-adder` with this contract, and if `make-adder` returns a value other than a function, then the server is to blame. If `make-adder` does return a function, but the resulting function is applied to a value other than an integer, then the client is to blame.

Similarly, the contract

```
(-> (-> integer? integer?) integer?)
```

describes functions that accept other functions as its input. If a server exports a function `twice` with this contract and the `twice` is applied to a value other than a function of one argument, then the client is to blame. If `twice` is applied to a function of one argument and `twice` calls the given function on a value other than an integer, then the server is to blame.

7.2.5 Contract Messages with "???"

You wrote your module. You added contracts. You put them into the interface so that client programmers have all the information from interfaces. It's a piece of art:

```
#lang racket

(provide/contract
 [deposit (-> (lambda (x)
                (and (number? x) (integer? x) (>= x 0)))
              any)])

(define this 0)
(define (deposit a) ...)
```

Several clients used your module. Others used their modules in turn. And all of a sudden one of them sees this error message:

> *bank-client broke the contract (-> ??? any) it had with myaccount on deposit; expected <???>, given: -10*

Clearly, `bank-client` is a module that uses `myaccount` but what is the *???* doing there? Wouldn't it be nice if we had a name for this class of data much like we have string, number, and so on?

For this situation, Racket provides *flat named contracts*. The use of "contract" in this term shows that contracts are first-class values. The "flat" means that the collection of data is a subset of the built-in atomic classes of data; they are described by a predicate that consumes all Racket values and produces a boolean. The "named" part says what we want to do, which is to name the contract so that error messages become intelligible:

```
#lang racket

(define (amount? x) (and (number? x) (integer? x) (>= x 0)))
(define amount (flat-named-contract 'amount amount?))

(provide/contract
 [deposit (amount . -> . any)])

(define this 0)
(define (deposit a) ...)
```

With this little change, the error message becomes all of the sudden quite readable:

bank-client broke the contract (-> amount any) it had with myaccount on deposit; expected <amount>, given: -10

7.3 Contracts on Functions in General

The `->` contract constructor works for functions that take a fixed number of arguments and where the result contract is independent of the input arguments. To support other kinds of functions, Racket supplies additional contract constructors, notably `->*` and `->i`.

7.3.1 Optional Arguments

Take a look at this excerpt from a string-processing module, inspired by the Scheme cookbook:

```
#lang racket

(provide/contract
 ; pad the given str left and right with
 ; the (optional) char so that it is centered
 [string-pad-center (->* (string? natural-number/c)
                         (char?)
                         string?)])

(define (string-pad-center str width [pad #\space])
  (define field-width (min width (string-length str)))
  (define rmargin (ceiling (/ (- width field-width) 2)))
  (define lmargin (floor (/ (- width field-width) 2)))
  (string-append (build-string lmargin (λ (x) pad))
                 str
                 (build-string rmargin (λ (x) pad))))
```

The module exports `string-pad-center`, a function that creates a string of a given `width` with the given string in the center. The default fill character is `#\space`; if the client module wishes to use a different character, it may call `string-pad-center` with a third argument, a `char`, overwriting the default.

The function definition uses optional arguments, which is appropriate for this kind of functionality. The interesting point here is the formulation of the contract for the `string-pad-center`.

The contract combinator `->*`, demands several groups of contracts:

- The first one is a parenthesized group of contracts for all required arguments. In this example, we see two: `string?` and `natural-number/c`.
- The second one is a parenthesized group of contracts for all optional arguments: `char?`.
- The last one is a single contract: the result of the function.

Note if a default value does not satisfy a contract, you won't get a contract error for this interface. If you can't trust yourself to get the initial value right, you need to communicate the initial value across a boundary.

7.3.2 Rest Arguments

The `max` operator consumes at least one real number, but it accepts any number of additional arguments. You can write other such functions using a "rest" argument, such as in `max-abs`:

See §4.4.1 "Declaring a Rest Argument" for an introduction to rest arguments.

```
(define (max-abs n . rst)
  (foldr (lambda (n m) (max (abs n) m)) (abs n) rst))
```

Describing this function through a contract requires a further extension of `->*`: a `#:rest` keyword specifies a contract on a list of arguments after the required and optional arguments:

```
(provide/contract
 [max-abs (->* (real?) () #:rest (listof real?) real?)])
```

As always for `->*`, the contracts for the required arguments are enclosed in the first pair of parentheses, which in this case is a single real number. The empty pair of parenthesis indicates that there are no optional arguments (not counting the rest arguments). The contract for the rest argument follows `#:rest`; since all additional arguments must be real numbers, the list of rest arguments must satisfy the contract `(listof real?)`.

7.3.3 Keyword Arguments

It turns out that the `->` contract constructor also contains support for keyword arguments. For example, consider this function, which creates a simple GUI and asks the user a yes-or-no question:

See §4.4.3 "Declaring Keyword Arguments" for an introduction to keyword arguments.

```
#lang racket/gui

(define (ask-yes-or-no-question question
                                #:default answer
```

```
                                #:title title
                                #:width w
                                #:height h)
  (define d (new dialog% [label title] [width w] [height h]))
  (define msg (new message% [label question] [parent d]))
  (define (yes) (set! answer #t) (send d show #f))
  (define (no) (set! answer #f) (send d show #f))
  (define yes-b (new button%
                     [label "Yes"] [parent d]
                     [callback (λ (x y) (yes))]
                     [style (if answer '(border) '())]))
  (define no-b (new button%
                    [label "No"] [parent d]
                    [callback (λ (x y) (no))]
                    [style (if answer '() '(border))]))
  (send d show #t)
  answer)

(provide/contract
 [ask-yes-or-no-question
  (-> string?
      #:default boolean?
      #:title string?
      #:width exact-integer?
      #:height exact-integer?
      boolean?)])
```

The contract for `ask-yes-or-no-question` uses `->`, and in the same way that `lambda` (or `define`-based functions) allows a keyword to precede a functions formal argument, `->` allows a keyword to precede a function contract's argument contract. In this case, the contract says that `ask-yes-or-no-question` must receive four keyword arguments, one for each of the keywords `#:default`, `#:title`, `#:width`, and `#:height`. As in a function definition, the order of the keywords in `->` relative to each other does not matter for clients of the function; only the relative order of argument contracts without keywords matters.

If you really want to ask a yes-or-no question via a GUI, you should use `message-box/custom`. For that matter, it's usually better to provide buttons with more specific answers than "yes" and "no."

7.3.4 Optional Keyword Arguments

Of course, many of the parameters in `ask-yes-or-no-question` (from the previous question) have reasonable defaults and should be made optional:

```
(define (ask-yes-or-no-question question
                                #:default answer
                                #:title [title "Yes or No?"]
                                #:width [w 400]
```

```
                                  #:height [h 200])
  ...)
```

To specify this function's contract, we need to use `->*` again. It supports keywords just as you might expect in both the optional and mandatory argument sections. In this case, we have the mandatory keyword `#:default` and optional keywords `#:title`, `#:width`, and `#:height`. So, we write the contract like this:

```
(provide/contract
 [ask-yes-or-no-question
  (->* (string?
        #:default boolean?)

       (#:title string?
        #:width exact-integer?
        #:height exact-integer?)

       boolean?)])
```

That is, we put the mandatory keywords in the first section, and we put the optional ones in the second section.

7.3.5 Contracts for `case-lambda`

A function defined with `case-lambda` might impose different constraints on its arguments depending on how many are provided. For example, a `report-cost` function might convert either a pair of numbers or a string into a new string:

See §4.4.4 "Arity-Sensitive Functions: `case-lambda`" for an introduction to `case-lambda`.

```
(define report-cost
  (case-lambda
    [(lo hi) (format "between $~a and $~a" lo hi)]
    [(desc) (format "~a of dollars" desc)]))
```

```
> (report-cost 5 8)
"between $5 and $8"
> (report-cost "millions")
"millions of dollars"
```

The contract for such a function is formed with the `case->` combinator, which combines as many functional contracts as needed:

```
(provide/contract
```

```
[report-cost
 (case->
  (integer? integer? . -> . string?)
  (string? . -> . string?))])
```

As you can see, the contract for `report-cost` combines two function contracts, which is just as many clauses as the explanation of its functionality required.

7.3.6 Argument and Result Dependencies

The following is an excerpt from an imaginary numerics module:

```
(provide/contract
 [real-sqrt (->i ([argument (>=/c 1)])
                 [result (argument) (<=/c argument)])])
```

The contract for the exported function `real-sqrt` uses the `->i` rather than `->*` function contract. The "i" stands for an *indy dependent* contract, meaning the contract for the function range depends on the value of the argument. The appearance of `argument` in the line for `result`'s contract means that the result depends on the argument. In this particular case, the argument of `real-sqrt` is greater or equal to 1, so a very basic correctness check is that the result is smaller than the argument.

In general, a dependent function contract looks just like the more general `->*` contract, but with names added that can be used elsewhere in the contract.

Going back to the bank-account example, suppose that we generalize the module to support multiple accounts and that we also include a withdrawal operation. The improved bank-account module includes a `account` structure type and the following functions:

```
(provide/contract
 [balance (-> account? amount/c)]
 [withdraw (-> account? amount/c account?)]
 [deposit (-> account? amount/c account?)])
```

Besides requiring that a client provide a valid amount for a withdrawal, however, the amount should be less than the specified account's balance, and the resulting account will have less money than it started with. Similarly, the module might promise that a deposit produces an account with money added to the account. The following implementation enforces those constraints and guarantees through contracts:

```
#lang racket

; section 1: the contract definitions
```

```
(struct account (balance))
(define amount/c natural-number/c)

; section 2: the exports
(provide/contract
 [create   (amount/c . -> . account?)]
 [balance  (account? . -> . amount/c)]
 [withdraw (->i ([acc account?]
                 [amt (acc) (and/c amount/c (<=/c (balance acc)))])
                [result (acc amt)
                        (and/c account?
                               (lambda (res)
                                 (>= (balance res)
                                     (- (balance acc) amt))))])]
 [deposit  (->i ([acc account?]
                 [amt amount/c])
                [result (acc amt)
                        (and/c account?
                               (lambda (res)
                                 (>= (balance res)
                                     (+ (balance acc) amt))))])])

; section 3: the function definitions
(define balance account-balance)

(define (create amt) (account amt))

(define (withdraw a amt)
  (account (- (account-balance a) amt)))

(define (deposit a amt)
  (account (+ (account-balance a) amt)))
```

The contracts in section 2 provide typical type-like guarantees for `create` and `balance`. For `withdraw` and `deposit`, however, the contracts check and guarantee the more complicated constraints on `balance` and `deposit`. The contract on the second argument to `withdraw` uses `(balance acc)` to check whether the supplied withdrawal amount is small enough, where `acc` is the name given within `->i` to the function's first argument. The contract on the result of `withdraw` uses both `acc` and `amt` to guarantee that no more than that requested amount was withdrawn. The contract on `deposit` similarly uses `acc` and `amount` in the result contract to guarantee that at least as much money as provided was deposited into the account.

As written above, when a contract check fails, the error message is not great. The following revision uses `flat-named-contract` within a helper function `mk-account-contract` to

provide better error messages.

```
#lang racket

; section 1: the contract definitions
(struct account (balance))
(define amount/c natural-number/c)

(define msg> "account a with balance larger than ~a expected")
(define msg< "account a with balance less than ~a expected")

(define (mk-account-contract acc amt op msg)
  (define balance0 (balance acc))
  (define (ctr a)
    (and (account? a) (op balance0 (balance a))))
  (flat-named-contract (format msg balance0) ctr))

; section 2: the exports
(provide/contract
 [create   (amount/c . -> . account?)]
 [balance  (account? . -> . amount/c)]
 [withdraw (->i ([acc account?]
                 [amt (acc) (and/c amount/c (<=/c (balance acc)))])
                [result (acc amt) (mk-account-
contract acc amt >= msg>)])]
 [deposit  (->i ([acc account?]
                 [amt amount/c])
                [result (acc amt)
                        (mk-account-contract acc amt <= msg<)])])

; section 3: the function definitions
(define balance account-balance)

(define (create amt) (account amt))

(define (withdraw a amt)
  (account (- (account-balance a) amt)))

(define (deposit a amt)
  (account (+ (account-balance a) amt)))
```

7.3.7 Checking State Changes

The `->i` contract combinator can also ensure that a function only modifies state according to certain constraints. For example, consider this contract (it is a slightly simplified from the

function `preferences:add-panel` in the framework):

```
(->i ([parent (is-a?/c area-container-window<%>)])
      [_ (parent)
       (let ([old-children (send parent get-children)])
         (λ (child)
           (andmap eq?
                   (append old-children (list child))
                   (send parent get-children))))])
```

It says that the function accepts a single argument, named `parent`, and that `parent` must be an object matching the interface `area-container-window<%>`.

The range contract ensures that the function only modifies the children of `parent` by adding a new child to the front of the list. It accomplishes this by using the `_` instead of a normal identifier, which tells the contract library that the range contract does not depend on the values of any of the results, and thus the contract library evaluates the expression following the `_` when the function is called, instead of when it returns. Therefore the call to the `get-children` method happens before the function under the contract is called. When the function under contract returns, its result is passed in as `child`, and the contract ensures that the children after the function return are the same as the children before the function called, but with one more child, at the front of the list.

To see the difference in a toy example that focuses on this point, consider this program

```
#lang racket
(define x '())
(define (get-x) x)
(define (f) (set! x (cons 'f x)))
(provide/contract
 [f (->i () [_ (begin (set! x (cons 'ctc x)) any/c)])]
 [get-x (-> (listof symbol?))])
```

If you were to require this module, call `f`, then the result of `get-x` would be `'(f ctc)`. In contrast, if the contract for `f` were

```
(->i () [res (begin (set! x (cons 'ctc x)) any/c)])
```

(only changing the underscore to `res`), then the result of `get-x` would be `'(ctc f)`.

7.3.8 Multiple Result Values

The function `split` consumes a list of `char`s and delivers the string that occurs before the first occurrence of `#\newline` (if any) and the rest of the list:

```
(define (split l)
  (define (split l w)
    (cond
      [(null? l) (values (list->string (reverse w)) '())]
      [(char=? #\newline (car l))
       (values (list->string (reverse w)) (cdr l))]
      [else (split (cdr l) (cons (car l) w))]))
  (split l '()))
```

It is a typical multiple-value function, returning two values by traversing a single list.

The contract for such a function can use the ordinary function arrow `->`, since `->` treats `values` specially when it appears as the last result:

```
(provide/contract
 [split (-> (listof char?)
            (values string? (listof char?)))])
```

The contract for such a function can also be written using `->*`:

```
(provide/contract
 [split (->* ((listof char?))
             ()
             (values string? (listof char?)))])
```

As before, the contract for the argument with `->*` is wrapped in an extra pair of parentheses (and must always be wrapped like that) and the empty pair of parentheses indicates that there are no optional arguments. The contracts for the results are inside `values`: a string and a list of characters.

Now, suppose that we also want to ensure that the first result of `split` is a prefix of the given word in list format. In that case, we need to use the `->i` contract combinator:

```
(define (substring-of? s)
  (flat-named-contract
    (format "substring of ~s" s)
```

```
    (lambda (s2)
      (and (string? s2)
           (<= (string-length s2) s)
           (equal? (substring s 0 (string-length s2)) s2)))))

(provide/contract
 [split (->i ([fl (listof char?)])
             (values [s (fl) (substring-of (list->string fl))]
                     [c (listof char?)]))])
```

Like `->*`, the `->i` combinator uses a function over the argument to create the range contracts. Yes, it doesn't just return one contract but as many as the function produces values: one contract per value. In this case, the second contract is the same as before, ensuring that the second result is a list of `char`s. In contrast, the first contract strengthens the old one so that the result is a prefix of the given word.

This contract is expensive to check, of course. Here is a slightly cheaper version:

```
(provide/contract
 [split (->i ([fl (listof char?)])
             (values [s (fl) (string-len/c (length fl))]
                     [c (listof char?)]))])
```

7.3.9 Fixed but Statically Unknown Arities

Imagine yourself writing a contract for a function that accepts some other function and a list of numbers that eventually applies the former to the latter. Unless the arity of the given function matches the length of the given list, your procedure is in trouble.

Consider this `n-step` function:

```
; (number ... -> (union #f number?)) (listof number) -> void
(define (n-step proc inits)
  (let ([inc (apply proc inits)])
    (when inc
      (n-step proc (map (λ (x) (+ x inc)) inits)))))
```

The argument of `n-step` is `proc`, a function `proc` whose results are either numbers or false, and a list. It then applies `proc` to the list `inits`. As long as `proc` returns a number, `n-step` treats that number as an increment for each of the numbers in `inits` and recurs. When `proc` returns `false`, the loop stops.

Here are two uses:

```
; nat -> nat
(define (f x)
  (printf "~s\n" x)
  (if (= x 0) #f -1))
(n-step f '(2))

; nat nat -> nat
(define (g x y)
  (define z (+ x y))
  (printf "~s\n" (list x y z))
  (if (= z 0) #f -1))

(n-step g '(1 1))
```

A contract for `n-step` must specify two aspects of `proc`'s behavior: its arity must include the number of elements in `inits`, and it must return either a number or `#f`. The latter is easy, the former is difficult. At first glance, this appears to suggest a contract that assigns a *variable-arity* to `proc`:

```
(->* ()
     (listof any/c)
     (or/c number? false/c))
```

This contract, however, says that the function must accept *any* number of arguments, not a *specific* but *undetermined* number. Thus, applying `n-step` to `(lambda (x) x)` and `(list 1)` breaks the contract because the given function accepts only one argument.

The correct contract uses the `unconstrained-domain->` combinator, which specifies only the range of a function, not its domain. It is then possible to combine this contract with an arity test to specify the correct `n-step`'s contract:

```
(provide/contract
 [n-step
  (->i ([proc (inits)
         (and/c (unconstrained-domain->
                 (or/c false/c number?))
                (λ (f) (procedure-arity-includes?
                        f
                        (length inits))))]
        [inits (listof number?)])
       ()
       any)])
```

7.4 Contracts: A Thorough Example

This section develops several different flavors of contracts for one and the same example: Racket's `argmax` function. According to its Racket documentation, the function consumes a procedure `proc` and a non-empty list of values, `lst`. It

> returns the *first* element in the list `lst` that maximizes the result of `proc`.

The emphasis on *first* is ours.

Examples:

```
> (argmax add1 (list 1 2 3))
3
> (argmax sqrt (list 0.4 0.9 0.16))
0.9
> (argmax second '((a 2) (b 3) (c 4) (d 1) (e 4)))
'(c 4)
```

Here is the simplest possible contract for this function:

version 1

```
#lang racket

(define (argmax f lov) ...)

(provide/contract
  [argmax (-> (-> any/c real?) (and/c pair? list?) any/c)])
```

This contract captures two essential conditions of the informal description of `argmax`:

- the given function must produce numbers that are comparable according to `<`. In particular, the contract `(-> any/c number?)` would not do, because `number?` also recognizes complex numbers in Racket.
- the given list must contain at least one item.

When combined with the name, the contract explains the behavior of `argmax` at the same level as an ML function type in a module signature (except for the non-empty list aspect).

Contracts may communicate significantly more than a type signature, however. Take a look at this second contract for `argmax`:

version 2

```
#lang racket

(define (argmax f lov) ...)

(provide/contract
  [argmax
    (->i ([f (-> any/c real?)] [lov (and/c pair? list?)]) ()
         (r (f lov)
            (lambda (r)
              (define f@r (f r))
              (for/and ((v lov)) (>= f@r (f v))))))])
```

It is a *dependent* contract that names the two arguments and uses the names to impose a predicate on the result. This predicate computes `(f r)` – where `r` is the result of `argmax` – and then validates that this value is greater than or equal to all values of `f` on the items of `lov`.

Is it possible that `argmax` could cheat by returning a random value that accidentally maximizes `f` over all elements of `lov`? With a contract, it is possible to rule out this possibility:

version 2 rev. a

```
#lang racket

(define (argmax f lov) ...)

(provide/contract
  [argmax
    (->i ([f (-> any/c real?)] [lov (and/c pair? list?)]) ()
         (r (f lov)
            (lambda (r)
              (define f@r (f r))
              (and
                (memq r lov)
                (for/and ((v lov)) (>= f@r (f v)))))))])
```

The `memq` function ensures that `r` is *intensionally equal* to one of the members of `lov`. Of course, a moment's worth of reflection shows that it is impossible to make up such a value. Functions are opaque values in Racket and without applying a function, it is impossible to determine whether some random input value produces an output value or triggers some exception. So we ignore this possibility from here on.

That is, "pointer equality" for those who prefer to think at the hardware level.

Version 2 formulates the overall sentiment of `argmax`'s documentation, but it fails to bring across that the result is the *first* element of the given list that maximizes the given function `f`. Here is a version that communicates this second aspect of the informal documentation:

version 3

```
#lang racket

(define (argmax f lov) ...)

(provide/contract
  [argmax
    (->i ([f (-> any/c real?)] [lov (and/c pair? list?)]) ()
         (r (f lov)
            (lambda (r)
              (define f@r (f r))
              (and (for/and ((v lov)) (>= f@r (f v)))
                   (eq? (first (memf (lambda (v) (= (f v) f@r)) lov))
                        r)))))])
```

That is, the `memf` function determines the first element of `lov` whose value under `f` is equal to `r`'s value under `f`. If this element is intensionally equal to `r`, the result of `argmax` is correct.

This second refinement step introduces two problems. First, both conditions recompute the values of `f` for all elements of `lov`. Second, the contract is now quite difficult to read. Contracts should have a concise formulation that a client can comprehend with a simple scan. Let us eliminate the readability problem with two auxiliary functions that have reasonably meaningful names:

version 3 rev. a

```
#lang racket

(define (argmax f lov) ...)

(provide/contract
  [argmax
    (->i ([f (-> any/c real?)] [lov (and/c pair? list?)]) ()
         (r (f lov)
            (lambda (r)
              (define f@r (f r))
              (and (is-first-max? r f@r f lov)
                   (dominates-all f@r f lov)))))])

; where

; f@r is greater or equal to all (f v) for v in lov
(define (dominates-all f@r f lov)
  (for/and ((v lov)) (>= (f v) f@r)))

; r is eq? to the first element v of lov for which (pred? v)
```

```
(define (is-first-max? r f@r f lov)
  (eq? (first (memf (lambda (v) (= (f v) f@r)) lov)) r))
```

The names of the two predicates express their functionality and, in principle, render it unnecessary to read their definitions.

This step leaves us with the problem of the newly introduced inefficiency. To avoid the recomputation of `(f v)` for all `v` on `lov`, we change the contract so that it computes these values and reuses them as needed:

version 3 rev. b

```
#lang racket

(define (argmax f lov) ...)

(provide/contract
  [argmax
    (->i ([f (-> any/c real?)] [lov (and/c pair? list?)]) ()
         (r (f lov)
            (lambda (r)
              (define f@r (f r))
              (define flov (map f lov))
              (and (is-first-max? r f@r (map list lov flov))
                   (dominates-all f@r flov)))))])

; where

; f@r is greater or equal to all f@v in flov
(define (dominates-all f@r flov)
  (for/and ((f@v flov)) (>= f@r f@v)))

; r is (second x) for the first x in flov+lov s.t. (= (first x)
f@r)
(define (is-first-max? r f@r lov+flov)
  (define fst (first lov+flov))
  (if (= (second fst) f@r)
      (eq? (first fst) r)
      (is-first-max? f@r r (rest lov+flov))))
```

Now the predicate on the result once again computes all values of `f` for elements of `lov` once.

The word "eager" comes from the literature on the linguistics of contracts.

Version 3 may still be too eager when it comes to calling `f`. While Racket's `argmax` always calls `f` no matter how many items `lov` contains, let us imagine for illustrative purposes that our own implementation first checks whether the list is a singleton. If so, the first element

would be the only element of `lov` and in that case there would be no need to compute `(f r)`. As a matter of fact, since `f` may diverge or raise an exception for some inputs, `argmax` should avoid calling `f` when possible.

The following contract demonstrates how a higher-order dependent contract needs to be adjusted so as to avoid being over-eager:

version 4

```
#lang racket

(define (argmax f lov)
  (if (empty? (rest lov))
      (first lov)
      ...))

(provide/contract
  [argmax
    (->i ([f (-> any/c real?)] [lov (and/c pair? list?)]) ()
         (r (f lov)
            (lambda (r)
              (cond
                [(empty? (rest lov)) (eq? (first lov) r)]
                [else
                 (define f@r (f r))
                 (define flov (map f lov))
                 (and (is-first-max? r f@r (map list lov flov))
                      (dominates-all f@r flov))])))))])

; where

; f@r is greater or equal to all f@v in flov
(define (dominates-all f@r lov) ...)

; r is (second x) for the first x in flov+lov s.t. (= (first x)
f@r)
(define (is-first-max? r f@r lov+flov) ...)
```

The `argmax` of Racket implicitly argues that it not only promises the first value that maximizes `f` over `lov` but also that `f` produces/produced a value for the result.

Note that such considerations don't apply to the world of first-order contracts. Only a higher-order (or lazy) language forces the programmer to express contracts with such precision.

The problem of diverging or exception-raising functions should alert the reader to the even more general problem of functions with side-effects. If the given function `f` has visible effects – say it logs its calls to a file – then the clients of `argmax` will be able to observe two sets of logs for each call to `argmax`. To be precise, if the list of values contains more than one element, the log will contain two calls of `f` per value on `lov`. If `f` is expensive to compute, doubling the calls imposes a high cost.

To avoid this cost and to signal problems with overly eager contracts, a contract system could record the i/o of contracted function arguments and use these hashtables in the dependency specification. This is a topic of on-going research in PLT. Stay tuned.

7.5 Contracts on Structures

Modules deal with structures in two ways. First they export `struct` definitions, i.e., the ability to create structs of a certain kind, to access their fields, to modify them, and to distinguish structs of this kind against every other kind of value in the world. Second, on occasion a module exports a specific struct and wishes to promise that its fields contain values of a certain kind. This section explains how to protect structs with contracts for both uses.

7.5.1 Guarantees for a Specific Value

If your module defines a variable to be a structure, then you can specify the structure's shape using `struct/c`:

```
#lang racket
(require lang/posn)

(define origin (make-posn 0 0))

(provide/contract
 [origin (struct/c posn zero? zero?)])
```

In this example, the module imports a library for representing positions, which exports a `posn` structure. One of the `posn`s it creates and exports stands for the origin, i.e., `(0,0)`, of the grid.

See also `vector/c` and similar contract combinators for (flat) compound data.

7.5.2 Guarantees for All Values

The book *How to Design Programs* teaches that `posn`s should contain only numbers in their two fields. With contracts we would enforce this informal data definition as follows:

```
#lang racket
(struct posn (x y))

(provide/contract
 [struct posn ((x number?) (y number?))]
 [p-okay posn?]
 [p-sick posn?])
```

```
(define p-okay (posn 10 20))
(define p-sick (posn 'a 'b))
```

This module exports the entire structure definition: `posn`, `posn?`, `posn-x`, `posn-y`, `set-posn-x!`, and `set-posn-y!`. Each function enforces or promises that the two fields of a `posn` structure are numbers — when the values flow across the module boundary. Thus, if a client calls `posn` on `10` and `'a`, the contract system signals a contract violation.

The creation of `p-sick` inside of the `posn` module, however, does not violate the contracts. The function `posn` is used internally, so `'a` and `'b` don't cross the module boundary. Similarly, when `p-sick` crosses the boundary of `posn`, the contract promises a `posn?` and nothing else. In particular, this check does *not* require that the fields of `p-sick` are numbers.

The association of contract checking with module boundaries implies that `p-okay` and `p-sick` look alike from a client's perspective until the client extracts the pieces:

```
#lang racket
(require lang/posn)

... (posn-x p-sick) ...
```

Using `posn-x` is the only way the client can find out what a `posn` contains in the `x` field. The application of `posn-x` sends `p-sick` back into the `posn` module and the result value – `'a` here – back to the client, again across the module boundary. At this very point, the contract system discovers that a promise is broken. Specifically, `posn-x` doesn't return a number but a symbol and is therefore blamed.

This specific example shows that the explanation for a contract violation doesn't always pinpoint the source of the error. The good news is that the error is located in the `posn` module. The bad news is that the explanation is misleading. Although it is true that `posn-x` produced a symbol instead of a number, it is the fault of the programmer who created a `posn` from symbols, i.e., the programmer who added

```
(define p-sick (posn 'a 'b))
```

to the module. So, when you are looking for bugs based on contract violations, keep this example in mind.

If we want to fix the contract for `p-sick` so that the error is caught when `sick` is exported, a single change suffices:

```
(provide/contract
 ...
 [p-sick (struct/c posn number? number?)])
```

That is, instead of exporting `p-sick` as a plain `posn?`, we use a `struct/c` contract to enforce constraints on its components.

7.5.3 Checking Properties of Data Structures

Contracts written using `struct/c` immediately check the fields of the data structure, but sometimes this can have disastrous effects on the performance of a program that does not, itself, inspect the entire data structure.

As an example, consider the binary search tree search algorithm. A binary search tree is like a binary tree, except that the numbers are organized in the tree to make searching the tree fast. In particular, for each interior node in the tree, all of the numbers in the left subtree are smaller than the number in the node, and all of the numbers in the right subtree are larger than the number in the node.

We can implement a search function `in?` that takes advantage of the structure of the binary search tree.

```
#lang racket

(struct node (val left right))

; determines if ‘n’ is in the binary search tree ‘b’,
; exploiting the binary search tree invariant
(define (in? n b)
  (cond
    [(null? b) #f]
    [else (cond
            [(= n (node-val b))
             #t]
            [(< n (node-val b))
             (in? n (node-left b))]
            [(> n (node-val b))
             (in? n (node-right b))])]))

; a predicate that identifies binary search trees
(define (bst-between? b low high)
  (or (null? b)
      (and (<= low (node-val b) high)
           (bst-between? (node-left b) low (node-val b))
           (bst-between? (node-right b) (node-val b) high))))

(define (bst? b) (bst-between? b -inf.0 +inf.0))
```

```
(provide (struct node (val left right)))
(provide/contract
 [bst? (any/c . -> . boolean?)]
 [in? (number? bst? . -> . boolean?)])
```

In a full binary search tree, this means that the `in?` function only has to explore a logarithmic number of nodes.

The contract on `in?` guarantees that its input is a binary search tree. But a little careful thought reveals that this contract defeats the purpose of the binary search tree algorithm. In particular, consider the inner `cond` in the `in?` function. This is where the `in?` function gets its speed: it avoids searching an entire subtree at each recursive call. Now compare that to the `bst-between?` function. In the case that it returns `#t`, it traverses the entire tree, meaning that the speedup of `in?` is lost.

In order to fix that, we can employ a new strategy for checking the binary search tree contract. In particular, if we only checked the contract on the nodes that `in?` looks at, we can still guarantee that the tree is at least partially well-formed, but without changing the complexity.

To do that, we need to use `define-contract-struct` in place of `struct`. Like `struct` (and more like `define-struct`), `define-contract-struct` defines a maker, predicate, and selectors for a new structure. Unlike `define-struct`, it also defines contract combinators, in this case `node/c` and `node/dc`. Also unlike `define-struct`, it does not allow mutators, making its structs always immutable.

The `node/c` function accepts a contract for each field of the struct and returns a contract on the struct. More interestingly, the syntactic form `node/dc` allows us to write dependent contracts, i.e., contracts where some of the contracts on the fields depend on the values of other fields. We can use this to define the binary search tree contract:

```
#lang racket

(define-contract-struct node (val left right))

; determines if ‘n’ is in the binary search tree ‘b’
(define (in? n b) ... as before ...)

; bst-between : number number -> contract
; builds a contract for binary search trees
; whose values are between low and high
(define (bst-between/c low high)
  (or/c null?
        (node/dc [val (between/c low high)]
                 [left (val) (bst-between/c low val)]
                 [right (val) (bst-between/c val high)])))
```

```
(define bst/c (bst-between/c -inf.0 +inf.0))

(provide make-node node-left node-right node-val node?)
(provide/contract
 [bst/c contract?]
 [in? (number? bst/c . -> . boolean?)])
```

In general, each use of `node/dc` must name the fields and then specify contracts for each field. In the above, the `val` field is a contract that accepts values between `low` and `high`. The `left` and `right` fields are dependent on the value of the `val` field, indicated by their second sub-expressions. Their contracts are built by recursive calls to the `bst-between/c` function. Taken together, this contract ensures the same thing that the `bst-between?` function checked in the original example, but here the checking only happens as `in?` explores the tree.

Although this contract improves the performance of `in?`, restoring it to the logarithmic behavior that the contract-less version had, it is still imposes a fairly large constant overhead. So, the contract library also provides `define-opt/c` that brings down that constant factor by optimizing its body. Its shape is just like the `define` above. It expects its body to be a contract and then optimizes that contract.

```
(define-opt/c (bst-between/c low high)
  (or/c null?
        (node/dc [val (between/c low high)]
                 [left (val) (bst-between/c low val)]
                 [right (val) (bst-between/c val high)])))
```

7.6 Abstract Contracts using `#:exists` and `#:∃`

The contract system provides existential contracts that can protect abstractions, ensuring that clients of your module cannot depend on the precise representation choices you make for your data structures.

The `provide/contract` form allows you to write

```
#:∃ name-of-a-new-contract
```

as one of its clauses. This declaration introduces the variable *`name-of-a-new-contract`*, binding it to a new contract that hides information about the values it protects.

You can type `#:exists` instead of `#:∃` if you cannot easily type unicode characters; in DrRacket, typing `\exists` followed by either alt-\ or control-\ (depending on your platform) will produce ∃.

As an example, consider this (simple) implementation of a stack datastructure:

```
#lang racket
```

```
(define empty '())
(define (enq top queue) (append queue (list top)))
(define (next queue) (car queue))
(define (deq queue) (cdr queue))
(define (empty? queue) (null? queue))

(provide/contract
 [empty (listof integer?)]
 [enq (-> integer? (listof integer?) (listof integer?))]
 [next (-> (listof integer?) integer?)]
 [deq (-> (listof integer?) (listof integer?))]
 [empty? (-> (listof integer?) boolean?)])
```

This code implements a queue purely in terms of lists, meaning that clients of this data structure might use `car` and `cdr` directly on the data structure (perhaps accidentally) and thus any change in the representation (say to a more efficient representation that supports amortized constant time enqueue and dequeue operations) might break client code.

To ensure that the stack representation is abstract, we can use `#:∃` in the `provide/contract` expression, like this:

```
(provide/contract
 #:∃ stack
 [empty stack]
 [enq (-> integer? stack stack)]
 [next (-> stack integer?)]
 [deq (-> stack (listof integer?))]
 [empty? (-> stack boolean?)])
```

Now, if clients of the data structure try to use `car` and `cdr`, they receive an error, rather than mucking about with the internals of the queues.

See also §7.8.2 “Exists Contracts and Predicates”.

7.7 Additional Examples

This section illustrates the current state of Racket’s contract implementation with a series of examples from *Design by Contract, by Example* [Mitchell02].

Mitchell and McKim’s principles for design by contract DbC are derived from the 1970s style algebraic specifications. The overall goal of DbC is to specify the constructors of an algebra in terms of its observers. While we reformulate Mitchell and McKim’s terminology and we use a mostly applicative approach, we retain their terminology of “classes” and “objects”:

- **Separate queries from commands.**

  A *query* returns a result but does not change the observable properties of an object. A *command* changes the visible properties of an object, but does not return a result. In applicative implementation a command typically returns an new object of the same class.

- **Separate basic queries from derived queries.**

  A *derived query* returns a result that is computable in terms of basic queries.

- **For each derived query, write a post-condition contract that specifies the result in terms of the basic queries.**

- **For each command, write a post-condition contract that specifies the changes to the observable properties in terms of the basic queries.**

- **For each query and command, decide on a suitable pre-condition contract.**

Each of the following sections corresponds to a chapter in Mitchell and McKim's book (but not all chapters show up here). We recommend that you read the contracts first (near the end of the first modules), then the implementation (in the first modules), and then the test module (at the end of each section).

Mitchell and McKim use Eiffel as the underlying programming language and employ a conventional imperative programming style. Our long-term goal is to transliterate their examples into applicative Racket, structure-oriented imperative Racket, and Racket's class system.

Note: To mimic Mitchell and McKim's informal notion of parametericity (parametric polymorphism), we use first-class contracts. At several places, this use of first-class contracts improves on Mitchell and McKim's design (see comments in interfaces).

7.7.1 A Customer-Manager Component

This first module contains some struct definitions in a separate module in order to better track bugs.

```
#lang racket
; data definitions

(define id? symbol?)
(define id-equal? eq?)
(define-struct basic-customer (id name address) #:mutable)

; interface
(provide/contract
```

```
 [id?                   (-> any/c boolean?)]
 [id-equal?             (-> id? id? boolean?)]
 [struct basic-customer ((id id?)
                         (name string?)
                         (address string?))])
; end of interface
```

This module contains the program that uses the above.

```
#lang racket

(require "1.rkt") ; the module just above

; implementation
; [listof (list basic-customer? secret-info)]
(define all '())

(define (find c)
  (define (has-c-as-key p)
    (id-equal? (basic-customer-id (car p)) c))
  (define x (filter has-c-as-key all))
  (if (pair? x) (car x) x))

(define (active? c)
  (define f (find c))
  (pair? (find c)))

(define not-active? (compose not active? basic-customer-id))

(define count 0)

(define (add c)
  (set! all (cons (list c 'secret) all))
  (set! count (+ count 1)))

(define (name id)
  (define bc-with-id (find id))
  (basic-customer-name (car bc-with-id)))

(define (set-name id name)
  (define bc-with-id (find id))
  (set-basic-customer-name! (car bc-with-id) name))

(define c0 0)
; end of implementation
```

```
(provide/contract
 ; how many customers are in the db?
 [count    natural-number/c]
 ; is the customer with this id active?
 [active?  (-> id? boolean?)]
 ; what is the name of the customer with this id?
 [name     (-> (and/c id? active?) string?)]
 ; change the name of the customer with this id
 [set-name (->d ([id id?] [nn string?])
                ()
                [result any/c] ; result contract
                #:post-cond
                (string=? (name id) nn))]

 [add      (->d ([bc (and/c basic-customer? not-active?)])
                ()
                ; A pre-post condition contract must use
                ; a side-effect to express this contract
                ; via post-conditions
                #:pre-cond (set! c0 count)
                [result any/c] ; result contract
                #:post-cond (> count c0))])
```

The tests:

```
#lang racket
(require rackunit rackunit/text-ui "1.rkt" "1b.rkt")

(add (make-basic-customer 'mf "matthias" "brookstone"))
(add (make-basic-customer 'rf "robby" "beverly hills park"))
(add (make-basic-customer 'fl "matthew" "pepper clouds town"))
(add (make-basic-customer 'sk "shriram" "i city"))

(run-tests
 (test-suite
  "manager"
  (test-equal? "id lookup" "matthias" (name 'mf))
  (test-equal? "count" 4 count)
  (test-true "active?" (active? 'mf))
  (test-false "active? 2" (active? 'kk))
  (test-true "set name" (void? (set-name 'mf "matt")))))
```

7.7.2 A Parameteric (Simple) Stack

```
#lang racket
```

```
; a contract utility
(define (eq/c x) (lambda (y) (eq? x y)))

(define-struct stack (list p? eq))

(define (initialize p? eq) (make-stack '() p? eq))
(define (push s x)
  (make-stack (cons x (stack-list s)) (stack-p? s) (stack-eq s)))
(define (item-at s i) (list-ref (reverse (stack-list s)) (- i 1)))
(define (count s) (length  (stack-list s)))
(define (is-empty? s) (null? (stack-list s)))
(define not-empty? (compose not is-empty?))
(define (pop s) (make-stack (cdr (stack-list s))
                            (stack-p? s)
                            (stack-eq s)))
(define (top s) (car (stack-list s)))

(provide/contract
 ; predicate
 [stack?     (-> any/c boolean?)]

 ; primitive queries
 ; how many items are on the stack?
 [count      (-> stack? natural-number/c)]

 ; which item is at the given position?
 [item-at
  (->d ([s stack?] [i (and/c positive? (<=/c (count s)))])
       ()
       [result (stack-p? s)])]

 ; derived queries
 ; is the stack empty?
 [is-empty?
  (->d ([s stack?])
       ()
       [result (eq/c (= (count s) 0))])]

 ; which item is at the top of the stack
 [top
  (->d ([s (and/c stack? not-empty?)])
       ()
       [t (stack-p? s)] ; a stack item, t is its name
       #:post-cond
       ([stack-eq s] t (item-at s (count s))))]
```

```
; creation
[initialize
 (->d ([p contract?] [s (p p . -> . boolean?)])
      ()
      ; Mitchell and McKim use (= (count s) 0) here to express
      ; the post-condition in terms of a primitive query
      [result (and/c stack? is-empty?)])]

; commands
; add an item to the top of the stack
[push
 (->d ([s stack?] [x (stack-p? s)])
      ()
      [sn stack?] ; result kind
      #:post-cond
      (and (= (+ (count s) 1) (count sn))
           ([stack-eq s] x (top sn))))]

; remove the item at the top of the stack
[pop
 (->d ([s (and/c stack? not-empty?)])
      ()
      [sn stack?] ; result kind
      #:post-cond
      (= (- (count s) 1) (count sn)))])
```

The tests:

```
#lang racket
(require rackunit rackunit/text-ui "2.rkt")

(define s0 (initialize (flat-contract integer?) =))
(define s2 (push (push s0 2) 1))

(run-tests
 (test-suite
  "stack"
  (test-true
   "empty"
   (is-empty? (initialize (flat-contract integer?) =)))
  (test-true "push" (stack? s2))
  (test-true
   "push exn"
   (with-handlers ([exn:fail:contract? (lambda _ #t)])
     (push (initialize (flat-contract integer?)) 'a)
```

```
    #f))
  (test-true "pop" (stack? (pop s2)))
  (test-equal? "top" (top s2) 1)
  (test-equal? "toppop" (top (pop s2)) 2)))
```

7.7.3 A Dictionary

```
#lang racket

; a shorthand for use below
(define-syntax ⇒
  (syntax-rules ()
    [(⇒ antecedent consequent) (if antecedent consequent #t)]))

; implementation
(define-struct dictionary (l value? eq?))
; the keys should probably be another parameter (exercise)

(define (initialize p eq) (make-dictionary '() p eq))
(define (put d k v)
  (make-dictionary (cons (cons k v) (dictionary-l d))
                   (dictionary-value? d)
                   (dictionary-eq? d)))
(define (rem d k)
  (make-dictionary
   (let loop ([l (dictionary-l d)])
     (cond
       [(null? l) l]
       [(eq? (caar l) k) (loop (cdr l))]
       [else (cons (car l) (loop (cdr l)))]))
   (dictionary-value? d)
   (dictionary-eq? d)))
(define (count d) (length (dictionary-l d)))
(define (value-for d k) (cdr (assq k (dictionary-l d))))
(define (has? d k) (pair? (assq k (dictionary-l d))))
(define (not-has? d) (lambda (k) (not (has? d k))))
; end of implementation

; interface
(provide/contract
 ; predicates
 [dictionary? (-> any/c boolean?)]
 ; basic queries
 ; how many items are in the dictionary?
```

```
 [count       (-> dictionary? natural-number/c)]
 ; does the dictionary define key k?
 [has?        (->d ([d dictionary?] [k symbol?])
                   ()
                   [result boolean?]
                   #:post-cond
                   ((zero? (count d)) . ⇒ . (not result)))]
 ; what is the value of key k in this dictionary?
 [value-for   (->d ([d dictionary?]
                    [k (and/c symbol? (lambda (k) (has? d k)))])
                   ()
                   [result (dictionary-value? d)])]
 ; initialization
 ; post condition: for all k in symbol, (has? d k) is false.
 [initialize  (->d ([p contract?] [eq (p p . -> . boolean?)])
                   ()
                   [result (and/c dictionary? (compose zero? count))])]
 ; commands
 ; Mitchell and McKim say that put shouldn't consume Void (null ptr)

 ; for v. We allow the client to specify a contract for all values
 ; via initialize. We could do the same via a key? parameter
 ; (exercise). add key k with value v to this dictionary
 [put         (->d ([d dictionary?]
                    [k (and symbol? (not-has? d))]
                    [v (dictionary-value? d)])
                   ()
                   [result dictionary?]
                   #:post-cond
                   (and (has? result k)
                        (= (count d) (- (count result) 1))
                        ([dictionary-eq? d] (value-
for result k) v)))]
 ; remove key k from this dictionary
 [rem         (->d ([d dictionary?]
                    [k (and/c symbol? (lambda (k) (has? d k)))])
                   ()
                   [result (and/c dictionary? not-has?)]
                   #:post-cond
                   (= (count d) (+ (count result) 1)))])
; end of interface
```

The tests:

```
#lang racket
(require rackunit rackunit/text-ui "3.rkt")
```

```
(define d0 (initialize (flat-contract integer?) =))
(define d (put (put (put d0 'a 2) 'b 2) 'c 1))

(run-tests
 (test-suite
  "dictionaries"
  (test-equal? "value for" 2 (value-for d 'b))
  (test-false "has?" (has? (rem d 'b) 'b))
  (test-equal? "count" 3 (count d))))
```

7.7.4 A Queue

```
#lang racket

; Note: this queue doesn't implement the capacity restriction
; of Mitchell and McKim's queue but this is easy to add.

; a contract utility
(define (all-but-last l) (reverse (cdr (reverse l))))
(define (eq/c x) (lambda (y) (eq? x y)))

; implementation
(define-struct queue (list p? eq))

(define (initialize p? eq) (make-queue '() p? eq))
(define items queue-list)
(define (put q x)
  (make-queue (append (queue-list q) (list x))
              (queue-p? q)
              (queue-eq q)))
(define (count s) (length  (queue-list s)))
(define (is-empty? s) (null? (queue-list s)))
(define not-empty? (compose not is-empty?))
(define (rem s)
  (make-queue (cdr (queue-list s))
              (queue-p? s)
              (queue-eq s)))
(define (head s) (car (queue-list s)))

; interface
(provide/contract
 ; predicate
 [queue?     (-> any/c boolean?)]
```

```
 ; primitive queries
 ; Imagine providing this 'query' for the interface of the module
 ; only. Then in Racket there is no reason to have count or is-
empty?
 ; around (other than providing it to clients). After all items is
 ; exactly as cheap as count.
 [items      (->d ([q queue?]) () [result (listof (queue-p? q))])]

 ; derived queries
 [count      (->d ([q queue?])
                  ; We could express this second part of the post
                  ; condition even if count were a module "at-
tribute"
                  ; in the language of Eiffel; indeed it would use
the
                  ; exact same syntax (minus the arrow and domain).
                  ()
                  [result (and/c natural-number/c
                                 (=/c (length (items q))))])]

 [is-empty?  (->d ([q queue?])
                  ()
                  [result (and/c boolean?
                                 (eq/c (null? (items q))))])]

 [head       (->d ([q (and/c queue? (compose not is-empty?))])
                  ()
                  [result (and/c (queue-p? q)
                                 (eq/c (car (items q))))])]
 ; creation
 [initialize (-> contract?
                 (contract? contract? . -> . boolean?)
                 (and/c queue? (compose null? items)))]

 ; commands
 [put        (->d ([oldq queue?] [i (queue-p? oldq)])
                  ()
                  [result
                   (and/c
                    queue?
                    (lambda (q)
                      (define old-items (items oldq))
                      (equal? (items q) (append old-
items (list i))))))])]
```

```
 [rem        (->d ([oldq (and/c queue? (compose not is-empty?))])
                  ()
                  [result
                   (and/c queue?
                          (lambda (q)
                            (equal? (cdr (items oldq)) (items q))))])])
; end of interface
```

The tests:

```
#lang racket
(require rackunit rackunit/text-ui "5.rkt")

(define s (put (put (initialize (flat-contract integer?) =) 2) 1))

(run-tests
 (test-suite
  "queue"
  (test-true
   "empty"
   (is-empty? (initialize (flat-contract integer?) =)))
  (test-true "put" (queue? s))
  (test-equal? "count" 2 (count s))
  (test-true "put exn"
             (with-handlers ([exn:fail:contract? (lambda _ #t)])
               (put (initialize (flat-contract integer?)) 'a)
               #f))
  (test-true "remove" (queue? (rem s)))
  (test-equal? "head" 2 (head s))))
```

7.8 Gotchas

7.8.1 Contracts and `eq?`

As a general rule, adding a contract to a program should either leave the behavior of the program unchanged, or should signal a contract violation. And this is almost true for Racket contracts, with one exception: `eq?`.

The `eq?` procedure is designed to be fast and does not provide much in the way of guarantees, except that if it returns true, it means that the two values behave identically in all respects. Internally, this is implemented as pointer equality at a low-level so it exposes information about how Racket is implemented (and how contracts are implemented).

Contracts interact poorly with `eq?` because function contract checking is implemented inter-

nally as wrapper functions. For example, consider this module:

```
#lang racket

(define (make-adder x)
  (if (= 1 x)
      add1
      (lambda (y) (+ x y))))
(provide/contract [make-adder (-> number? (-> number? number?))])
```

It exports the `make-adder` function that is the usual curried addition function, except that it returns Racket's `add1` when its input is `1`.

You might expect that

```
(eq? (make-adder 1)
     (make-adder 1))
```

would return `#t`, but it does not. If the contract were changed to `any/c` (or even `(-> number? any/c)`), then the `eq?` call would return `#t`.

Moral: Do not use `eq?` on values that have contracts.

7.8.2 Exists Contracts and Predicates

Much like the `eq?` example above, `#:∃` contracts can change the behavior of a program.

Specifically, the `null?` predicate (and many other predicates) return `#f` for `#:∃` contracts, and changing one of those contracts to `any/c` means that `null?` might now return `#t` instead, resulting in arbitrarily different behavior depending on this boolean might flow around in the program.

```
#lang racket/exists
```

To work around the above problem, the `racket/exists` library behaves just like the `racket`, but where predicates signal errors when given `#:∃` contracts.

Moral: Do not use predicates on `#:∃` contracts, but if you're not sure, use `racket/exists` to be safe.

7.8.3 Defining Recursive Contracts

When defining a self-referential contract, it is natural to use `define`. For example, one might try to write a contract on streams like this:

```
> (define stream/c
    (promise/c
     (or/c
      null?
      (cons/c number? stream/c))))
reference to undefined identifier: stream/c
```

Unfortunately, this does not work because the value of `stream/c` is needed before it is defined. Put another way, all of the combinators evaluate their arguments eagerly, even thought the values that they accept do not.

Instead, use

```
(define stream/c
 (promise/c
  (or/c
   null?
   (cons/c 1
           (recursive-contract stream/c)))))
```

The use of `recursive-contract` delays the evaluation of the identifier `stream/c` until after the contract is first checked, long enough to ensure that `stream/c` is defined.

See also §7.5.3 “Checking Properties of Data Structures”.

7.8.4 Mixing `set!` and `provide/contract`

The contract library assumes that variables exported via `provide/contract` are not assigned to, but does not enforce it. Accordingly, if you try to `set!` those variables, you may be surprised. Consider the following example:

```
> (module server racket
    (define (inc-x!) (set! x (+ x 1)))
    (define x 0)
    (provide/contract [inc-x! (-> void?)]
                      [x integer?]))
> (module client racket
    (require 'server)
```

```
    (define (print-latest) (printf "x is ~s\n" x))

    (print-latest)
    (inc-x!)
    (print-latest))
> (require 'client)
x is 0
x is 0
```

Both calls to `print-latest` print `0`, even though the value of `x` has been incremented (and the change is visible inside the module `x`).

To work around this, export accessor functions, rather than exporting the variable directly, like this:

```
#lang racket

(define (get-x) x)
(define (inc-x!) (set! x (+ x 1)))
(define x 0)
(provide/contract [inc-x! (-> void?)]
                  [get-x (-> integer?)])
```

Moral: This is a bug that we will address in a future release.