Chapter 24

Whole numbers

24.1 What is a whole number?

A *whole number*[1] is a non-negative integer: any of the numbers $0, 1, 2, 3, \ldots$.

What does the "..." in the above definition mean? It basically means "and so on," or "you know the rest." But what if you were talking to somebody who *really didn't* "know the rest"? Perhaps an alien from another planet, whose mathematical background was so different from yours that he/she/it couldn't fill in the "and so on". How would you define whole numbers to someone who didn't already know what whole numbers were?

In the 1880's, two mathematicians named Richard Dedekind and Giuseppe Peano addressed this problem more or less as follows:

- 0 is a whole number
- If α is a whole number, then so is $S(\alpha)$

The S was intended to stand for "successor" — for example, 1 is the successor of 0, and 2 is the successor of 1. However, the above definition doesn't require that you already know what 0, or 1, or 2, or "successor", or "plus" mean.

24.1.1 Defining wholes from structs

Imagine that Racket didn't already know about whole numbers. We could *define* them as follows:

```
; A built-whole is either 0 or (S built-whole).
(define-struct successor [previous])
(define (S x) ; shorter name for convenience
  (make-successor x))
(define (P x) ; shorter name for convenience
  (successor-previous x))
```

(I use the name *built-whole* to distinguish it from "ordinary wholes", which we'll use in the next section.)

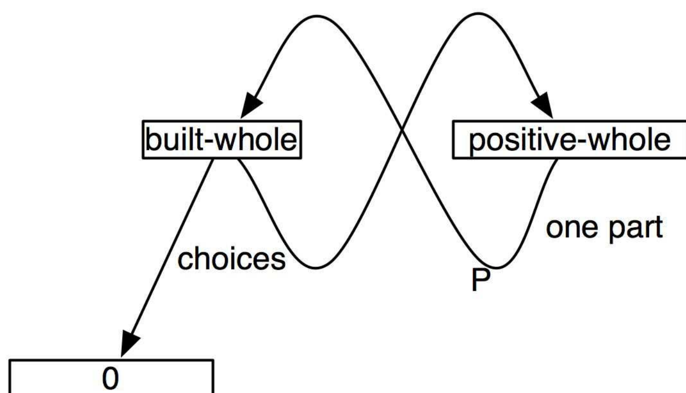
We would then start constructing examples of the data type:

[1] Actually, I was brought up to call these "natural numbers", and Racket includes a `natural?` function to tell whether something is one of them. But some books define "natural numbers" to start at 1 rather than 0. By contrast, everybody seems to agree that the "whole numbers" start at 0, so that's the term I'll use in this book.

- 0
- `(S 0)`, which "means" 1
- `(S (S 0))`, which "means" 2
- `(P (S (S 0)))`, another way to represent 1
- `(S (S (S (S (S 0)))))`, which "means" 5

and so on.

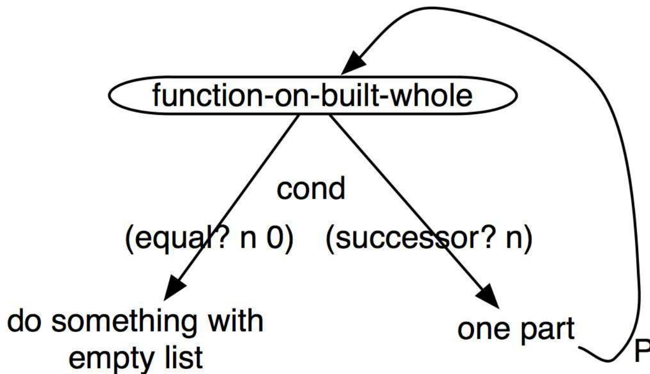
The above definition should remind you of the definition of a list in Chapter 22: a list is either `empty` or `(cons` *`object list`*`)`. We defined lists by choices; one of the choices had two parts (which we could get by using `first` and `rest`), one of which was itself a list.



This combination of definition by choices and by parts led us to a standard way to write functions on lists.

Following that analogy, how would one write functions on this *built-whole* data type? The data type is defined by two choices, one of which has one part, which is another built-whole. So the template (collapsed into a single function) looks like

```
(define (function-on-built-whole n)
  (cond [(equal? n 0) ...]
        [(successor? n)
         ; n                                successor
         ; (P n)                            built-whole
         ; (function-on-built-whole (P n)) whatever this returns
        ]))
```



Worked Exercise 24.1.1 ***Develop a function*** *`spams` that takes in a* built-whole *and returns a list with that many copies of the string `"spam"`.*

Solution: The contract is clearly

```
; spams :  built-whole -> list-of-string
```

Since the data type has two choices, we need to make sure we've got an example of each, and a more complex example

```
(check-expect (spams 0) empty)
(check-expect (spams (S 0)) (list "spam"))
(check-expect (spams (S (S (S 0)))) (list "spam" "spam" "spam"))
```

For the function skeleton, we'll start by copying the single-function template from above, changing the function name:

```
(define ( spams n)
  (cond [(equal?  n 0) ...]
        [(successor?  n)
         ; n               successor
         ; (P n)           built-whole
         ; ( spams (P n)) list of strings
        ]))
```

The answer in the 0 case is obviously `empty`. For the non-zero case, let's try an inventory with values:

```
(define (spams n)
  (cond [(equal?  n 0)    empty]
        [(successor?  n)
         ; n              successor       3
         ; (P n)          built-whole     2
         ; (spams (P n)) list of strings (list "spam" "spam")
         ; right answer   list of strings
                          ; (list "spam" "spam" "spam")
        ]))
```

The obvious way to get from `(list "spam" "spam")` to `(list "spam" "spam" "spam")` is by `cons`-ing on another `"spam"`, so ...

```
(define (spams n)
  (cond [(equal?  n 0)      empty]
        [(successor?  n)
         ; n             successor       3
         ; (P n)         built-whole     2
         ; (spams (P n)) list of strings (list "spam" "spam")
         ; right answer  list of strings
                         ; (list "spam" "spam" "spam")
         (cons "spam" (spams (P n)))
        ]))
```

Test this on the examples, and it should work. Make up some examples of your own; does it do what you expect? ■

SIDEBAR:

The word "spam" today means "commercial junk e-mail". Have you ever wondered how it got that meaning?

"Spam" was originally a brand name for a "Spiced Ham" product sold by the Hormel meat company. In 1970, the British TV show "Monty Python's Flying Circus" aired a comedy sketch about a restaurant that was peculiar in two ways: first, every item on its menu included Spam, and second, one table of the restaurant was occupied by Vikings who, on several occasions during the sketch, started singing "Spam, spam, spam, spam ..."

In 1970, there was no such thing as e-mail. By 1980, e-mail was a well-known phenomenon, although not many people had it, and comics could start joking "If we have electronic mail, pretty soon we'll have electronic junk mail." By 1990, it was no longer a joke but a nuisance. Legend has it that somebody (I don't know who or when — probably in the 1980's) was going through his/her inbox deleting junk mail and muttering "junk ...junk ...junk ..." when the Monty Python sketch popped into his/her head and (s)he replaced the word "junk" with the word "spam". And the rest is history.

Exercise 24.1.2 ***Develop a function*** *`copies` that takes in a string and a* built-whole*, and produces a list of that many copies of the string.*

Exercise 24.1.3 ***Develop a function*** *`whole-value` that takes in a* built-whole *and returns the ordinary number that it "means". For example,*

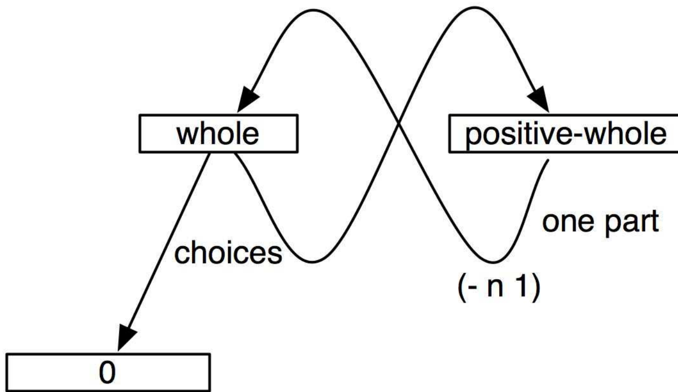
```
(check-expect (whole-value 0) 0)
(check-expect (whole-value (S 0)) 1)
(check-expect (whole-value (P (S (S (S (S (S 0))))))) 4)
```

24.1.2 Wholes, the way we really do it

One can do a lot with this definition of wholes, but writing `(S (S (S (S (S 0)))))` for 5 is a royal pain. In fact, Racket already knows about numbers, including whole numbers, so we can use Racket's predefined arithmetic operations.

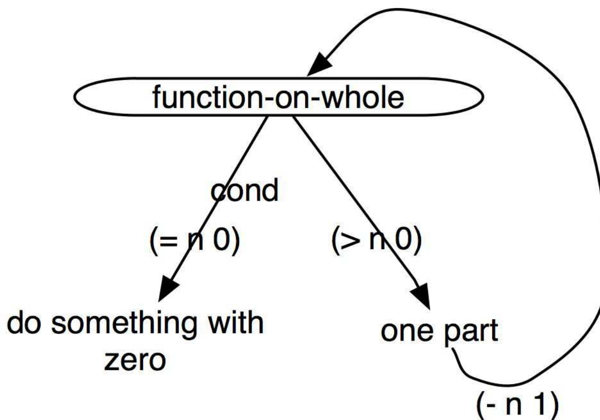
```
A whole is either 0 or (add1 whole).
```

We can replace
`(equal? n 0)` with the predefined `(zero? n)` or `(= n 0)`;
`(S n)` with the predefined `(add1 n)` or `(+ n 1)`;
`(P n)` with the predefined `(sub1 n)` or `(- n 1)`; and
`(successor? n)` with the predefined `(positive? n)` or `> n 0)`.



The resulting template looks like

```
(define (function-on-whole n)
  (cond [(= n 0) ...]
        [(> n 0)
         ; n                              positive whole
         ; (- n 1)                        whole
         ; (function-on-whole (- n 1)) whatever this returns
        ]))
```



Worked Exercise 24.1.4 ***Re-write*** *the* `spams` *function of Exercise 24.1.1 to work on ordinary, built-in whole numbers.*

Solution: The contract changes to take in an ordinary whole number:

```
; spams : whole-number -> list-of-strings
```

The examples change to use ordinary number notation:

```
(check-expect (spams 0) empty)
(check-expect (spams 1) (list "spam"))
(check-expect (spams 3) (list "spam" "spam" "spam"))
```

The function definition is exactly the same as before, but replacing the *built-whole* functions with standard Racket arithmetic functions:

```
(define (spams n)
  (cond [(= n 0)       empty]
        [(> n 0)
         ; n                positive whole 3
         ;  (- n 1)         whole          2
         ; (spams  (- n 1)) list of strings(list "spam" "spam")
         ; right answer     list of strings
                            ; (list "spam" "spam" "spam")
         (cons "spam" (spams (- n 1)))
        ]))
```

Try this and make sure it still works. ∎

Exercise 24.1.5 ***Re-write*** *the* `copies` *function of Exercise 24.1.2 to take in a string and an (ordinary) whole number.*

Exercise 24.1.6 ***Develop a function*** `count-down` *that takes in an (ordinary) whole number and produces a list of the whole numbers from it down to 0. For example,*

```
(check-expect (count-down 4) (list 4 3 2 1 0))
```

Exercise 24.1.7 ***Develop a function*** `add-up-to` *that takes in a whole number and returns the sum of all the whole numbers up to and including it.*

Hint: The formula $n(n+1)/2$ solves the same problem, so you can use it to check your answers. But you should write your function by actually adding up all the numbers, not by using this formula.

Exercise 24.1.8 ***Develop a function*** `factorial` *that takes in a whole number and returns the product of all the whole numbers from 1 up to and including it.*

Hint: What is the "right answer" for 0? There are at least two possible ways to answer this: you could decide that the function *has* no answer at 0 (so the base case is at 1, not 0), or you could pick an answer for 0 so that the other answers all come out right. Mathematicians generally choose the latter.

Exercise 24.1.9 ***Develop a function*** `fibonacci` *that takes in a whole number n and produces the n-th Fibonacci number. The Fibonacci numbers are defined as follows: the 0-th Fibonacci number is 0, the 1st Fibonacci number is 1, and each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers. For example,*

```
(check-expect (fibonacci 0) 0)
(check-expect (fibonacci 1) 1)
(check-expect (fibonacci 2) 1)
(check-expect (fibonacci 3) 2)
(check-expect (fibonacci 4) 3)
(check-expect (fibonacci 5) 5)
(check-expect (fibonacci 6) 8)
(check-expect (fibonacci 7) 13)
```

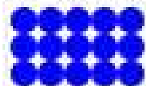
Hint: The usual template involves calling `(fibonacci (- n 1))` inside the body of `fibonacci`. In this case, you'll probably want to call `(fibonacci (- n 2))` as well. However, that doesn't make sense unless you know that `(- n 2)` *is* a whole number, so your base case needs to handle both 0 and 1.

Note: The definition of `fibonacci` that you get by following the template for whole numbers is correct, but extremely slow. On my computer, `(fibonacci 30)` takes about a second; `(fibonacci 35)` takes about ten seconds; and `(fibonacci 37)` takes almost thirty seconds. Try watching it in the Stepper, and you'll see that it asks the same question over and over. See if you can find a way to solve the same problem much more efficiently, using a helper function with some extra parameters. We'll see another way to fix this problem in Section 30.3.

SIDEBAR:

"Fibonacci" is the modern name for Leonardo filius Bonacci ("son of Bonaccio"), a mathematician who lived in Pisa, Italy in the 12th and 13th centuries. He is best known today for this sequence of numbers, which has surprising applications in biology, number theory, architecture, *etc.* But his most significant impact on the world was probably persuading European scholars to switch from Roman numerals by showing how much easier it is to do arithmetic using Arabic numerals.

Exercise 24.1.10 ***Develop a function*** *named* `dot-grid` *(remember this from Chapter 5?) that takes two whole numbers* `width` *and* `height` *and produces a rectangular grid of circles with* `width` *columns and* `height` *rows.*



Exercise 24.1.11 ***Develop a function*** *named* `randoms` *that takes in two whole numbers* `how-many` *and* `limit` *and produces a list of* `how-many` *numbers, each chosen randomly from 0 up to* `limit`*.*

Hint: Use the template on `how-many`, not on `limit`.

Exercise 24.1.12 ***Develop a function*** *named* `random-posns` *that takes in three whole numbers* `how-many`*,* `max-x`*, and* `max-y` *and produces a list of* `how-many` *posns, each with x chosen randomly between 0 and* `max-x`*, and y chosen randomly between 0 and* `max-y`*.*

Exercise 24.1.13 ***Develop a function*** *named* `table-of-squares` *that takes in a whole number and returns a list of* `posn`*s representing a table of numbers and their squares from the given number down to 0. For example,*

```
(check-expect (table-of-squares 4)
  (list (make-posn 4 16)
        (make-posn 3 9)
        (make-posn 2 4)
        (make-posn 1 1)
        (make-posn 0 0)))
```

Note: I've put these in descending order because it's easier to write the function that way. It would be nice to produce the table in increasing order instead. We'll see how to do that in the next section.

24.2 Different base cases, different directions

Recall Exercise 24.1.7, a function that adds up the positive integers from a specified number down to 0. What if we wanted to add up the positive numbers from a specified number down to, say, 10 instead?

Worked Exercise 24.2.1 ***Develop a function*** `add-up-from-10` *that takes in a whole number $n \geq 10$ and returns the sum $10 + 11 + \ldots + n$.*

Generalize this *to a function* `add-up-between` *that takes in two whole numbers m and n and returns the sum $m + (m + 1) + \ldots + n$.*

Solution: The function takes in a "whole number $\geq$ 10", which is a new data type. Here's its data definition:

```
; A whole-num>=10 is either 10, or (add1 whole-num>=10)
```

The contract and examples are easy:

```
; add-up-from-10 :  whole-number>=10 -> number
(check-expect (add-up-from-10 10) 10)
(check-expect (add-up-from-10 11) 21)
(check-expect (add-up-from-10 15) 65)
```

Since we've changed the data type, we'll need a new template:

```
(define (function-on-wn>=10 n)
  (cond [(= n 10) ...]
        [(> n 10)
         ; n                             whole number > 10
         ; (- n 1)                       whole number >= 10
         ; (function-on-wn>=10 (- n 1)) whatever this returns
        ]))
```

With this, the definition is easy:

```
(define (add-up-from-10 n)
  (cond [(= n 10) 10]
        [(> n 10)
         ; n                          whole number > 10
         ; (- n 1)                    whole number >= 10
         ; (add-up-from-10 (- n 1)) number
         (+ n (add-up-from-10 (- n 1)))
        ]))
```

It feels a bit inelegant to have $n \geq 10$ be part of the contract; could we reasonably make the function work correctly on *all* whole numbers? We would have to choose a "right answer" for numbers less than 10. In that case, there are no numbers to add up, so the answer should be 0.

```
; add-up-from-10 :  whole-number -> number
(check-expect (add-up-from-10 8) 0)
(check-expect (add-up-from-10 10) 10)
(check-expect (add-up-from-10 11) 21)
(check-expect (add-up-from-10 15) 65)
(define (add-up-from-10 n)
  (cond [(< n 10) 0]
        [(= n 10) 10]
        [(> n 10)
         ; n                          whole number > 10
         ; (- n 1)                    whole number >= 10
         ; (add-up-from-10 (- n 1)) number
         (+ n (add-up-from-10 (- n 1)))
        ]))
```

This passes all its tests, but on further consideration, we realize that the right answer to the `(= n 10)` case is the same as 10 plus the right answer to the `(< n 10)` case; we could leave out the `(= n 10)` case, replacing `(> n 10)` with `(>= n 10)`, and it would *still* pass all its tests. **Try this** for yourself.

The more general `add-up-between` function differs from `add-up-from-10` only by replacing the 10 with the extra parameter m:

```
;  add-up-between :  whole-number(m) whole-number(n) -> number
(check-expect (add-up-between 8 6) 0)
(check-expect (add-up-between 8 8) 8)
(check-expect (add-up-between 7 9) 24)
(define (add-up-between  m n)
  (cond [(< n  m) 0]
        [(>= n  m)
         ; n                            whole number >  m
         ; (- n 1)                      whole number >=  m
         ; (add-up-between  m (- n 1))number
         (+ n (add-up-between  m (- n 1)))
        ]))
```

Exercise 24.2.2 ***Develop a function count-down-to*** *that takes in two whole numbers* `low` *and* `high` *and produces a list of the numbers* `high`*,* `high`*-1, ...* `low` *in that order. If* `low` $>$ `high`*, it should return an empty list.*

What if we wanted the list in *increasing* order? Rather than treating `high` as a "whole number $\geq$ `low`", and calling the function recursively on `(sub1 high)`, we instead treat `low` as a "whole number $\leq$ `high`", and calling the function recursively on `(add1 low)`.

Exercise 24.2.3 ***Develop a function count-up-to*** *that takes in two whole numbers* `low` *and* `high` *and produces a list of the numbers* `low`*,* `low`*+1, ...*`high`*. If* `low` $>$ `high`*, it should return an empty list.*

Exercise 24.2.4 ***Develop a function*** `increasing-table-of-squares` *which takes in a whole number n and returns a list of* `posn`*s representing a table of numbers and their squares from 0 up to the given number.*

24.3 Peano arithmetic

Imagine that for some reason the `+` function wasn't working correctly on your computer (although `add1` and `sub1` still worked). Could we make do without `+`?

It would be pretty easy to write a function to add 2:

```
; add2 :  number -> number
(check-expect (add2 0) 2)
(check-expect (add2 1) 3)
(check-expect (add2 27) 29)
(define (add2 x)
  (add1 (add1 x)))
```

But can we write a function that takes in *two* whole numbers as parameters and adds them?

Worked Exercise 24.3.1 ***Develop a function*** `wn-add` *to add two whole numbers, without using any built-in arithmetic operators except* `add1`*,* `sub1`*,* `zero?`*, and* `positive?`*.*

Solution: The contract and examples are straightforward:

```
; wn-add :  whole-num (m) whole-num (n) -> whole-num
(check-expect (wn-add 0 0) 0)
(check-expect (wn-add 0 1) 1)
(check-expect (wn-add 0 3) 3)
(check-expect (wn-add 1 0) 1)
(check-expect (wn-add 3 0) 3)
(check-expect (wn-add 3 8) 11)
```

We have *two* whole-number parameters. In Chapter 25, we'll discuss how to handle this sort of situation in general, but for now let's just follow the template on one of them, pretending the other one is simple:

```
(define (wn-add m n)
  (cond [(zero?  n) ...]
        [(positive?  n)
         ; m                      whole
         ; n                      positive whole
         ; (sub1 n)               whole
         ; (wn-add m (sub1 n)) whole
         ...
        ]))
```

Now we need to fill in the ... parts. The "zero" case is easy: if $n = 0$, then $m + n = m + 0 = m$. For the nonzero case, we'll do an inventory with values:

```
(define (wn-add m n)
  (cond [(zero?  n)  m]
        [(positive?  n)
         ; m                      whole          3
         ; n                      positive whole 8
         ; (sub1 n)               whole          7
         ; (wn-add m (sub1 n)) whole          10
         ; right answer           whole          11
         ...
        ]))
```

Remember that we can only use `add1` and `sub1`, not `+` or `-`. So the obvious way to get 11 from 10 is `add1`:

```
(define (wn-add m n)
  (cond [(zero?  n) m]
        [(positive?  n)
         ; m                      whole 3
         ; n                      positi8e whole
         ; (sub1 n)               whole 7
         ; (wn-add m (sub1 n)) whole 10
         ; right answer           whole 11
         (add1 (wn-add m (sub1 n)))
        ]))
```

Does this make sense? Is it always true that $m + n = 1 + (m + (n - 1))$? Of course; that's simple algebra. ∎

Exercise 24.3.2 ***Develop a function*** *`wn-mult` which multiplies two whole numbers together without using any built-in arithmetic operators except `add1`, `sub1`, `zero?`, and `positive?`. You* are *allowed to use `wn-add`, because it's not built-in; we just defined it from these operators.*

All the remaining exercises in this section are subject to the same restriction: "without using any built-in arithmetic operators except `add1`, `sub1`, `zero?`, and `positive?`." You may, of course, re-use the functions you've already written in this section.

Exercise 24.3.3 ***Develop a function*** *`wn-raise` which, given two whole numbers m and n, computes m^n.*

Exercise 24.3.4 ***Develop a function*** `wn<=` *which, given two whole numbers* m *and* n*, tells whether* $m \leq n$*.*

Exercise 24.3.5 ***Develop a function*** `wn=` *which, given two whole numbers* m *and* n*, tells whether they're equal or not.*

Exercise 24.3.6 ***Develop a function*** `wn-sub` *which, given two whole numbers* m *and* n*, computes* $m - n$*.*

Hint: In this chapter, we've defined whole numbers, but not negative numbers, and we haven't promised that `sub1` works on anything other than a positive whole number. There are two ways you can write this function:

- The "purist" way uses `sub1` only on positive whole numbers, and produces an error message if you try to subtract a larger number from a smaller (this was actually a common mathematical practice in the 18th century)
- The "pragmatist" way relies on the fact that Racket *really does* know about negative numbers, and `sub1` *really does* work on any number, not only positive wholes. This way you can write `wn-sub` to work on *any* two whole numbers. The problem is that the result may not be a whole number, so code like `(wn-sub x (wn-sub y z))` may not work.

Exercise 24.3.7 ***Develop*** *two functions* `wn-quotient` *and* `wn-remainder` *which, given two whole numbers* m *and* n*, compute the quotient and remainder of dividing* m *by* n*. Both should produce an error message if* $n = 0$*.*

Exercise 24.3.8 ***Develop a function*** `wn-prime?` *which tells whether a given whole number is prime.*

Hint: There are several ways to do this. One way is to define a helper function `not-divisible-up-to?` which, given two whole numbers m and n, tells whether m is "not divisible by anything up to n" (except of course 1).

Racket also knows about *fractions*, but if it didn't, we could define them ourselves, just as we've defined wholes, addition, multiplication, and so on.

Exercise 24.3.9 ***Define a struct*** `frac` *that represents a fraction in terms of whole numbers (as we've defined them).*

Exercise 24.3.10 ***Develop a function*** `frac=` *that takes in two* `frac` *objects and tells whether they're equal. (**Careful!** What does it mean for two fractions to be equal?)*

Exercise 24.3.11 ***Develop a function*** `reduce-frac` *that takes in a* `frac` *and returns an equivalent* `frac` *in lowest terms,* i.e. *with no common factors between numerator and denominator.*

Exercise 24.3.12 ***Develop a function*** `frac-mult` *that takes in two* `frac`*s and returns their product, as a* `frac`*.*

Exercise 24.3.13 ***Develop a function*** `frac-add` *that takes in two* `fracs` *and returns their sum, as a* `frac`*.*

24.4 The wholes in binary

Dedekind and Peano's definition of the wholes isn't the only way to define them. Here's another approach.

Almost every computer built since about 1950 has used *binary* or *base-two* notation to represent numbers: for example, the number 19 is written 10011, indicating $1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$. In decimal notation, it's really easy to multiply by 10 (just write a 0 at the end of the number). Similarly, in base two, it's really easy to multiply by 2 (just write a 0 at the end of the number). This inspires the following data definition:

A binary-whole-number is either

- 0, or
- $S_0(whole)$, or
- $S_1(whole)$

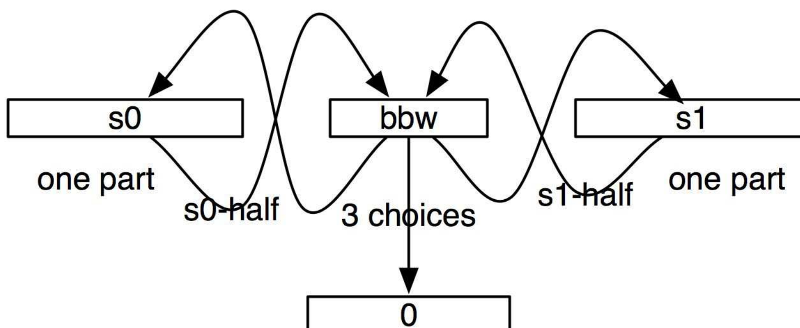
where $S_0(x)$ is intended to correspond to writing a 0 at the end of the number x, and $S_1(x)$ to writing a 1 at the end of x.

24.4.1 Defining binary wholes from structs

Let's try this in Racket.

```
A built-binary-whole is either
0,
(S0 built-binary-whole), or
(S1 built-binary-whole).

(define-struct s0 (half))
(define-struct s1 (half))
(define (S0 x) (make-s0 x)) ; for short
(define (S1 x) (make-s1 x)) ; for short
```



Some examples of this data type are

- 0
- `(S1 0)`, which "means" 1
- `(S0 (S1 0))`, which "means" 2
- `(S1 (S1 0))`, which "means" 3
- `(S1 (S1 (S0 (S0 (S1 0)))))`, which "means" 19

Obviously, it's a lot easier to write 19 as
`(S1 (S1 (S0 (S0 (S1 0)))))`
than as
`(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S 0))))))))))))))))))),`
the way we would have written 19 in section 24.1.1.

A template based on this data definition has three cases: is it zero, is it an `s0` structure, or is it an `s1` structure? I'll use *bbw* as shorthand for *built-binary-whole*.

```
(define (function-on-bbw n)
  (cond [(equal?  n 0) ...]
        [(s0?  n)
         ; (s0-half n)                      bbw
         ; (function-on-bbw (s0-half n)) whatever
         ...]
        [(s1?  n)
         ; (s1-half n)                      bbw
         ; (function-on-bbw (s1-half n)) whatever
         ...]
        ))
```

Worked Exercise 24.4.1 *Re-write the* `spams` *function to take in a* built-binary-whole.

Solution: The contract and examples are straightforward:

```
; spams :  built-binary-whole -> string-list
(check-expect (spams 0) empty)
(check-expect (spams (S1 0)) (list "spam"))
(check-expect (spams (S0 (S1 0))) (list "spam" "spam"))
(check-expect (spams (S0 (S1 (S1 0))))
  (list "spam" "spam" "spam" "spam" "spam" "spam"))
(check-expect (spams (S1 (S1 (S1 0))))
  (list "spam" "spam" "spam" "spam" "spam" "spam" "spam"))
```

The template gives us

```
(define ( spams n)
  (cond [(equal?  n 0) ...]
        [(s0?  n)
         ; (s0-half n)         whole
         ; ( spams (s0-half n))string-list
         ...]
        [(s1?  n)
         ; (s1-half n)         whole
         ; ( spams (s1-half n))string-list
         ...]
        ))
```

Obviously, the right answer to the zero case is `empty`. For the other cases, we'll use an inventory with values.

```
(define (spams n)
  (cond [(equal?  n 0)  empty]
        [(s0?  n)
         ; n              whole       (S0 (S1 (S1 0))), i.e. 6
         ; (s0-half n)  whole       (S1 (S1 0)), i.e. 3
         ; (spams (s0-half n))
         ;                string-list (list "spam" "spam "spam")
         ; right answer string-list
         ; (list "spam" "spam" "spam" "spam" "spam" "spam")
         ...]
        [(s1?  n)
         ; n              whole       (S1 (S1 (S1 0))), i.e. 7
         ; (s1-half n)  whole       (S1 (S1 0)), i.e. 3
         ; (spams (s1-half n))
         ;                string-list (list "spam" "spam" "spam")
         ; right answer string-list
         ; (list "spam" "spam" "spam" "spam" "spam" "spam "spam")
         ...]
        ))
```

Now, how can you get a list of 6 spams from a list of 3 spams? There are a number of ways, but the most obvious one is to append two copies of it together. Which seems appropriate, since the recursive call is supposed to return "half as many" spams.

How to get a list of 7 spams from a list of 3 spams? Since the recursive call is on "half" of an odd number, it's really $(n-1)/2$. So to get n from $(n-1)/2$, you make two copies and add one more. The function definition becomes

```
(define (spams n)
  (cond [(equal?  n 0) empty]
        [(s0?  n)
         ; n            whole       (S0 (S1 (S1 0))), i.e. 6
         ; (s0-half n)  whole       (S1 (S1 0)), i.e. 3
         ; (spams (s0-half n))
         ;              string-list (list "spam" "spam "spam")
         ; right answer string-list
         ; (list "spam" "spam" "spam" "spam" "spam" "spam")
         (append (spams (s0-half n))
                 (spams (s0-half n))) ]
        [(s1?  n)
         ; n            whole       (S1 (S1 (S1 0))), i.e. 7
         ; (s1-half n)  whole       (S1 (S1 0)), i.e. 3
         ; (spams (s1-half n))
         ;              string-list (list "spam" "spam" "spam")
         ; right answer string-list
         ; (list "spam" "spam" "spam" "spam" "spam" "spam "spam")
         (cons "spam" (append (spams (s1-half n))
                              (spams (s1-half n)))) ]
        ))
```

■

Exercise 24.4.2 ***Rewrite*** *the* `copies` *function to take in a* built-binary-whole*.*

Exercise 24.4.3 ***Develop a function*** `binary-add1` *that takes in a* built-binary-whole *and returns the next larger* built-binary-whole*. For example, the next larger whole after 5 is 6, so*

```
(check-expect (binary-add1 (S1 (S0 (S1 0)))) (S0 (S1 (S1 0))))
```

Exercise 24.4.4 ***Develop a function*** `bbw-value` *that takes in a* built-binary-whole *and returns the (ordinary) whole number that it represents. For example,*

```
(check-expect (binary-whole-value 0) 0)
(check-expect (binary-whole-value (S1 0)) 1)
(check-expect (binary-whole-value (S0 (S1 (S1 (S0 (S1 0)))))) 22)
```

24.4.2 Binary whole numbers, the way we really do it

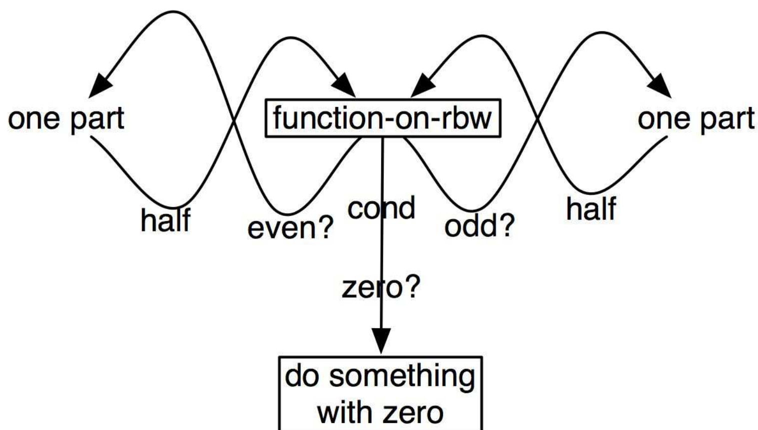
Again, Racket already knows about numbers and arithmetic, so instead of the structures `s0` and `s1`, we might use

```
(define (S0 x) (* x 2))
(define (S1 x) (+ 1 (* x 2)))
(define (half x) (quotient x 2))
```

plus the built-in functions `zero?`, `even?`, and `odd?`.

The template becomes (using *rbw* as shorthand for "real binary whole")

```
#|
(define (function-on-rbw n)
  (cond [(zero?  n) ...]
        [(even?  n)
         ; (half n)                          rbw
         ; (function-on-rbw (half n))        whatever
         ...]
        [(odd?  n)
         ; (half n)                          rbw
         ; (function-on-rbw (half n))        whatever
         ...]
        ))
|#
```



Worked Exercise 24.4.5 ***Re-write*** *the* `spams` *function to take in an ordinary whole number, using the binary template.*

Solution: The contract and examples are the same as in Exercise 24.1.4. The definition becomes

```
(define (binary-spams n)
  (cond [( zero?  n) empty]
        [( even?  n)
         ; n              whole       6
         ; ( half n)      whole       3
         ; (binary-spams ( half n))
         ;                string-list (list "spam" "spam "spam")
         ; right answer   string-list
         ; (list "spam" "spam" "spam" "spam" "spam" "spam")
         (append (binary-spams ( half n))
                 (binary-spams ( half n))) ]
        [( odd?  n)
         ; n              whole       7
         ; ( half n)      whole       3
         ; (binary-spams ( half n))
         ;                string-list (list "spam" "spam" "spam")
         ; right answer   string-list
         ; (list "spam" "spam" "spam" "spam" "spam" "spam "spam")
         (cons "spam" (append (binary-spams ( half n))
                              (binary-spams ( half n)))) ]
        ))
```

■

Exercise 24.4.6 ***Re-write*** *the* `copies` *function of Exercise 24.4.2 so that it takes in an ordinary whole number, but is still written using the binary template.*

Exercise 24.4.7 ***Re-write*** *the* `binary-add1` *function of Exercise 24.4.3 so that it takes in an ordinary whole number, but is still written using the binary template rather than calling the built-in* `add1` *or* `+`*. For example,*

```
(check-expect (binary-add1 5) 6)
```

Exercise 24.4.8 ***Re-write*** *the* `dot-grid` *function of Exercise 24.1.10 by using the binary template.*

Exercise 24.4.9 ***Re-write*** *the* `randoms` *function of Exercise 24.1.11 by using the binary template.*

Exercise 24.4.10 ***Re-write*** *the* `random-posns` *function of Exercise 24.1.12 by using the binary template.*

Exercise 24.4.11 ***Essay question:*** *Notice that I've picked* some *of the functions from Section 24.1 to re-do using the binary template. Why did I pick these and not the others? What kind of function lends itself to solving with the binary template, and what kind doesn't?*

24.5 Review of important words and concepts

Programmers often want a computer to do something a specified number of times. If the "number of times" is driven by a list of data, we can use the techniques of Chapters 22 and 23, but if it really is just a number, as in `copies` or `dot-grid`, we can use the analogous technique of *whole-number recursion*.

Racket, like most other programming languages, has built-in support for arithmetic on whole numbers and other kinds of numbers. (Most languages don't handle fractions, or very large whole numbers, as well as Racket does, but that's a separate issue.) However, in this chapter we've shown how to *define* arithmetic, whole numbers, and fractions. Along the way, we've learned a useful technique for writing functions that do things a specified number of times, like `copies` or `dot-grid`.

The whole numbers can be defined either using "successor", as Dedekind and Peano did, or using binary notation, as most modern computers do.

24.6 Reference: Built-in functions on whole numbers

In this chapter, we've introduced the built-in functions

- `zero?`
- `positive?`
- `sub1`