4: Structures

Compound data

The teaching languages provide a general mechanism called **structures**.

They permit the “bundling” of several values into one.

In many situations, data is naturally grouped, and most programming languages provide some mechanism to do this.

Extended example: points

We can create a type of structure to hold two coordinates:

```
(define-struct point (x y))
```

This does not create any structures, but sets up the ability to create `point` structures with two fields named `x` and `y`.

The `define-struct` expression provides functions to:

- create a `point` structure holding two values
- test whether something is a `point` structure
- access the values within a `point` structure

Working with point structures

We create a point with a **constructor** function: `(make-point 3 4)`

A point is a value, and we can treat it like other values we have seen.

```
(define p (make-point 3 4))
```

We access a value bundled in a point with an **accessor** function.

```
(point-x p) ⇒ 3
(point-y p) ⇒ 4
```

The **type predicate** `point?` tests for point-ness.

```
(point? p) ⇒ true
(point? 17) ⇒ false
```

Structures in the substitution model

An expression such as `(make-point 8 1)` is considered a value.

This expression will not be rewritten by the Stepper or our semantic rules.

The expression `(make-point (+ 4 4) (- 3 2))` would be rewritten to (eventually) yield `(make-point 8 1)`.

Here are the rewrite rules needed:

`(point-x (make-point vx vy))` ⇒ `vx`

`(point-y (make-point vx vy))` ⇒ `vy`

`(point? (make-point vx vy))` ⇒ `true`

`(point? v)` ⇒ `false` for other `v`

Functions that consume point structures

```
; distance: Point Point -> Number
; computes the Euclidean distance between p1 and p2
; Example:
   (check-expect (distance (make-point 1 1)
                           (make-point 4 5))
                 5)

(define (distance p1 p2)
 (sqrt (+ (sqr (- (point-x p1) (point-x p2)))
          (sqr (- (point-y p1) (point-y p2))))))
```

Functions that produce point structures

```
; scale: Point Number -> Point
; scales the point by the given factor
; Example:
   (check-expect (scale (make-point 3 4) 0.5)
                 (make-point 1.5 2))

(define (scale p factor)
 (make-point (* factor (point-x p))
             (* factor (point-y p))))
```

Tracing a use of scale

```
(scale (make-point 3 4) 0.5)
⇒  (make-point (* 0.5 (point-x (make-point 3 4)))
               (* 0.5 (point-y (make-point 3 4))))
⇒  (make-point (* 0.5 3)
               (* 0.5 (point-y (make-point 3 4))))
⇒  (make-point 1.5
                (* 0.5 (point-y (make-point 3 4))))
⇒  (make-point 1.5 (* 0.5 4))
⇒  (make-point 1.5 2)
```

Misusing point structures

Although we have been using numbers as coordinates, `make-point` can accept any two Racket values.

```
(distance (make-point 'Spider 'Man)
          (make-point 'Peter 'Parker))
```

This causes a run-time error only when the subtraction function in the body of `distance` tries to subtract `'Peter` from `'Spider`.

We address this with a **data definition**.

```
; A Point is a (make-point n m),
;   where n and m are Numbers.
```

Templates

We introduce another element of the design recipe.

A **template** is a code skeleton based on a data definition, to aid in writing the body of functions that consume that type of data.

Data Definition:

- A Point `p` is a `(make-point n m)`, where `n` and `m` are Numbers.

From this, we can see that `n` is `(point-x p)`, and `m` is `(point-y p)`. We put these into a template.

```
; my-point-fn : Point -> ?
(define (my-point-fn p)
  ... (point-x p) ...
  ... (point-y p) ...)
```

We need to replace each ellipsis with code, or remove it. We may not need both the accessor expressions.

This template is pretty simple, but we will soon see more useful ones.

General semantics for structures

Consider a structure `s` with n fields.

```
(define-struct s (f1 ... fn))
```

This introduces a constructor function `make-s`, accessor functions `s-f1` through `s-fn`, and a type predicate `s?`, with the following rewrite rules:

`(make-s v1 ... vn)` is a value.

`(s-fi (make-s v1 ... vi ... vn))` ⇒ `vi` for all `i`.

`(s? (make-s v1 ... vn))` ⇒ `true`

`(s? v)` ⇒ `false` for other `v`.

Extended example: simulating natural numbers

Natural numbers are provided by Racket as part of the built-in number system.

But if they were not provided, we could simulate them using structures. We will spend some time exploring this idea.

This exercise gives us insight into the foundations of mathematics and into the close relationship between mathematics and computer science.

We will choose to use the following structure definitions:

```
(define-struct Z ())
(define-struct S (pred))
```

The `Z` structure has no fields.
We can make one with the expression `(make-Z)`.

The `S` structure has one field.
We will use it to hold either a `Z` structure or another `S` structure.

A note on the definition of natural numbers

You may be used to the natural numbers starting at 1.

But in computer science, and in formal logic and the foundations of mathematics, the natural numbers start at 0.

If we don't make this choice, we have to add special cases for 0, and everything gets messier.

Nat: a Racket representation of natural numbers

We will choose the following interpretations.

- The Racket expression `(make-Z)` represents the natural number 0.
- What an `S` structure represents is a bit more complicated. “S” in this case stands for “successor”. The successor of the natural number 0 is the natural number 1.

The Racket expression `(make-S (make-Z))` is our representation of the successor of 0, namely 1.

Similarly, `(make-S (make-S (make-Z)))` represents 2, because 2 is the successor of 1.

In general, if `e` is the representation of the natural number n, then `(make-S e)` is the representation of the natural number $n+1$.

A data definition for our representation of natural numbers

We will call our representation of natural numbers **Nat**.

Data definition:

A Nat is either `(make-Z)` or it is `(make-S p)`, where `p` is a Nat.

This data definition is different from the one we had for Point. It is **self-referential**: it contains a reference to the thing being defined.

In computer science, we call this a **recursive** definition.

It is not a circular definition.

Part of it, referring to `(make-Z)`, is not self-referential.

The self-referential part refers to how to create a new Nat value from an already-justified Nat value.

Computing the predecessor

The predecessor of a natural number n is $n - 1$.
Zero has no predecessor.

How might we write a Racket function to compute the predecessor of a Nat?

The contract will be

```
; predecessor : Nat -> Nat
```

Since a Nat is either a `Z` or an `S` structure, this suggests the following skeleton:

```
(define (pred nat)
  (cond
    [(Z? nat) ...]
    [(S? nat) ...]))
```

Computing the predecessor

```
(define (pred nat)
  (cond
    [(Z? nat) ...]
    [(S? nat) ...]))
```

Filling in the ellipses is straightforward.

```
(define (pred nat)
  (cond
    [(Z? nat) (error "can't apply pred to Z")]
    [(S? nat) (S-pred nat)]))


(check-error (pred (make-Z)) "can't apply pred to Z")
(check-expect (pred (make-S (make-Z)))
              (make-Z))
(check-expect (pred (make-S (make-S (make-Z))))
              (make-S (make-Z)))
```

A possible template for Nat

The previous example suggests the following template for functions that consume a Nat. It combines the idea of the Point template with a `cond` for the two cases in the Nat definition.

```
(define (my-nat-fn nat)
  (cond
    [(Z? nat) ...]
    [(S? nat) ...]))
```

A possible template for Nat

The previous example suggests the following template for functions that consume a Nat. It combines the idea of the Point template with a `cond` for the two cases in the Nat definition.

```
(define (my-nat-fn nat)
  (cond
    [(Z? nat) ...]
    [(S? nat) ... (S-pred nat) ...]))
```

This will work for some functions, but the next example shows that in some cases, it would help to be a little more precise.

Adding two Nats

Let’s try to work out addition for Nats.

```
; plus : Nat Nat -> Nat
```

The template only mentions one parameter. We can add a second one.

```
(define (plus nat1 nat2)
  (cond
    [(Z? nat1) ... nat2 ...]
    [(S? nat1) ... (S-pred nat1) ... nat2 ...]))
```

Filling in the first case is easy.

```
(define (plus nat1 nat2)
  (cond
    [(Z? nat1) nat2]
    [(S? nat1) ... (S-pred nat1) ... nat2 ...]))
```

The second case seems harder.

Adding two Nats

```
(define (plus nat1 nat2)
  (cond
    [(Z? nat1) nat2]
    [(S? nat1) ... (S-pred nat1) ... nat2 ...]))
```

In the case where `nat1` is an `S` structure, what can we do with its predecessor and `nat2`?

We can add them together and compute the successor of the result.

```
(define (plus nat1 nat2)
  (cond
    [(Z? nat1) nat2]
    [(S? nat1) (make-S (plus (S-pred nat1) nat2))]))
```

This is a **recursive** function: it uses itself in its own body.

Like the recursive definition of Nat, it is not circular.

Traces will give us confidence that it works.

Tracing plus

```
(plus (make-Z) (make-Z))
⇒  (cond
      [(Z? (make-Z)) (make-Z)]
      [(S? (make-Z))
         (make-S (plus (S-pred (make-Z)) (make-Z)))])
⇒  (cond
      [true (make-Z)]
      [(S? (make-Z))
         (make-S (plus (S-pred (make-Z)) (make-Z)))])
⇒  (make-Z)
```

Tracing plus (continued)

```
(plus (make-S (make-Z)) (make-Z))
⇒  (cond
      [(Z? (make-S (make-Z))) (make-Z)]
      [(S? (make-S (make-Z)))
         (make-S (plus (S-pred (make-S (make-Z)))
         (make-Z)))])
⇒  (cond
      [false (make-Z)]
      [(S? (make-S (make-Z)))
         (make-S (plus (S-pred (make-S (make-Z)))
         (make-Z)))])
⇒  (cond
      [(S? (make-S (make-Z)))
         (make-S (plus (S-pred (make-S (make-Z)))
         (make-Z)))])
```

Tracing plus (continued)

```
⇒  (cond
      [(S? (make-S (make-Z)))
        (make-S (plus (S-pred (make-S (make-Z)))
        (make-Z)))])
⇒  (cond
      [true (make-S (plus (S-pred (make-S (make-Z)))
        (make-Z)))])
⇒  (make-S (plus (S-pred (make-S (make-Z))) (make-Z)))
⇒  (make-S (plus (make-Z) (make-Z)))
```

At this point, the trace is not done, but we have traced the evaluation of `(plus (make-Z) (make-Z))` already.

Condensed traces

It is clear that traces will be very long and boring if we list every step.

We introduce the symbol $\Rightarrow^*$ to mean “yields in zero or more steps”.

Typically we will use it to skip to the answer in a `cond`.

```
(plus (make-S (make-Z)) (make-Z))
⇒* (make-S (plus (make-Z) (make-Z)))
⇒* (make-S (make-Z))
```

We can now do a condensed trace of a longer computation.

```
(plus (make-S (make-S (make-Z)))
      (make-S (make-S (make-Z))))
```

Condensed traces (continued)

```
(plus (make-S (make-S (make-Z)))
      (make-S (make-S (make-Z))))
⇒* (make-S (plus (make-S (make-Z))
                 (make-S (make-S (make-Z)))))
⇒* (make-S (make-S (plus (make-Z)
                         (make-S (make-S (make-Z))))))
⇒* (make-S (make-S (make-S (make-S (make-Z)))))
```

We have demonstrated that two plus two equals four.

A template for a recursive function consuming a Nat

Recall the data definition for Nat:

A Nat is either `(make-Z)` or it is `(make-S p)`, where `p` is a Nat.

This suggests a refinement of our template:

```
(define (my-nat-fn nat)
  (cond
    [(Z? nat) ...]
    [(S? nat) ... (my-nat-fn (S-pred nat)) ...]))
```

The recursive application of the function happens in the case corresponding to the self-referential part of the data definition.

This is known as **pure structural recursion**.

Pure structural recursion

A function consuming Nats uses pure structural recursion if it conforms to the template we just developed.

We can add more parameters (as in the case of `plus`, where we added `nat2`) but they must be unchanged in the recursive application of the function within its body.

Pure structural recursion is to be preferred because it is easier to reason about, as we will see.

However, not every function consuming Nats uses pure structural recursion.

A function that does not use pure structural recursion

The following function computes the result of dividing a natural number by two and rounding down.

```
(define (idiv2 nat)
  (cond
    [(Z? nat) (make-Z)]
    [(equal? nat (make-S (make-Z))) (make-Z)]
    [else (make-S (idiv2 (S-pred (S-pred nat))))]))
```

Why is this not pure structural recursion?

- There is an extra case for the representation of 1.
- The recursive application is not on `(S-pred nat)`, but on `(S-pred (S-pred nat))`.

This function uses **generative recursion**.

Tracing idiv2

If we apply `idiv2` to the representation of 5, it should produce the representation of 2, and a condensed trace confirms this.

```
(idiv2 (make-S (make-S (make-S (make-S
                                    (make-S (make-Z)))))))
⇒* (make-S (idiv2 (make-S (make-S (make-S (make-Z))))))
⇒* (make-S (make-S (idiv2 (make-S (make-Z)))))
⇒* (make-S (make-S (make-Z)))
```

Make sure you can fill in the missing steps. This is a good idea in general when dealing with condensed traces.

Correctness of plus

How can we be sure that `plus` correctly implements addition?

How can you be sure that your idea of addition is actually correct?

If you were to try to write down a precise definition of natural number and a precise definition of addition, they would probably be longer and more complicated than the ones we just saw.

We can show that `plus` has the properties we expect of addition.

Some simple theorems

Theorem

$0 + 2 = 2$

This is almost too simple to be considered a mathematical theorem, but the corresponding theorem in our Racket representation is not so obviously true.

Theorem

```
(plus (make-Z) (make-S (make-S (make-Z))))
⇒* (make-S (make-S (make-Z)))
```

The proof of this second theorem is a trace.

A theorem about Racket computation

Theorem

```
(plus (make-Z) (make-S (make-S (make-Z))))
⇒* (make-S (make-S (make-Z)))
```

Proof

```
(plus (make-Z) (make-S (make-S (make-Z))))
⇒  (cond
      [(Z? (make-Z)) (make-S (make-S (make-Z)))]
      [(S? (make-Z)) (make-S (plus (S-pred (make-Z))
                                   (make-S (make-S (make-Z)))))])
⇒  (cond
      [true (make-S (make-S (make-Z)))]
      [(S? (make-Z)) (make-S (plus (S-pred (make-Z))
                                   (make-S (make-S (make-Z)))))])
⇒  (make-S (make-S (make-Z)))
```

Some more interesting theorems

Theorem

For all natural numbers n, $0 + n = n$.

Theorem

For all Nats `x`, `(plus (make-Z) x)` $\Rightarrow^*$ `x`.

Some more interesting theorems (Continued)

Theorem

For all Nats `x`, `(plus (make-Z) x)` $\Rightarrow^*$ `x`.

Proof

```
(plus (make-Z) x)
⇒  (cond
      [(Z? (make-Z)) x]
      [(S? (make-Z)) (make-S (plus (S-pred (make-Z))
                                   x))])
⇒  (cond
      [true x]
      [(S? (make-Z)) (make-S (plus (S-pred (make-Z))
                                   x))])
⇒  x
```

This is not a trace. It is a **trace schema**.
Substituting any specific Nat for `x` gives a valid trace.

Proving for-all statements

In general, to prove a “For all Nats `x`” statement, one method is to remove the for-all and prove the rest of the statement treating `x` as an unknown.

Our reasoning must remain valid when any specific Nat is substituted for `x` in the proof.

This is known as “for-all introduction” in formal logic.

It is quite common.

For example, in mathematics, when we prove $(x+y)(x-y) = x^2 - y^2$ by using the distributive law, we are really proving “For all x, y, $(x+y)(x-y) = x^2 - y^2$” using for-all introduction.

If our logical system permits this as valid reasoning, it is useful.

But this method has its limitations.

A situation where for-all introduction fails

Theorem

For all natural numbers n, $n + 0 = n$.

Theorem

For all Nats `x`, `(plus x (make-Z))` $\Rightarrow^*$ `x`.

Trying to construct a trace schema fails, because we don't know enough about `x`.

We can try a case analysis. Since `x` is a Nat, it is either `(make-Z)` or `(make-S y)` for some Nat `y`.

A situation where for-all introduction fails

Here is the case where `x` is `(make-Z)`.

Proof

```
(plus (make-Z) (make-Z))
⇒  (cond
      [(Z? (make-Z)) (make-Z)]
      [(S? (make-Z)) (make-S (plus (S-pred (make-Z))
                                   (make-Z)))])
⇒  (cond
      [true (make-Z)]
      [(S? (make-Z)) (make-S (plus (S-pred (make-Z))
                                   (make-Z)))])
⇒  (make-Z)
```

But the reasoning is not so clear in the case where `x` is `(make-S y)`.

A situation where for-all introduction fails (continued)

Here is the case where `x` is `(make-S y)`.

Proof

```
(plus (make-S y) (make-Z))
⇒  (cond
      [(Z? (make-S y)) (make-Z)]
      [(S? (make-S y)) (make-S (plus (S-pred (make-S y))
                                     (make-Z)))])
⇒  (cond
      [false (make-Z)]
      [(S? (make-S y)) (make-S (plus (S-pred (make-S y))
                                     (make-Z)))])
⇒  (cond
      [(S? (make-S y)) (make-S (plus (S-pred (make-S y))
                                     (make-Z)))])
```

A situation where for-all introduction fails (continued)

Proof Continued

```
⇒  (cond
      [(S? (make-S y)) (make-S (plus (S-pred (make-S y))
                                      (make-Z)))])
⇒  (cond
      [true (make-S (plus (S-pred (make-S y))
                          (make-Z)))])
⇒  (make-S (plus (S-pred (make-S y)) (make-Z)))
⇒  (make-S (plus y (make-Z)))
```

We have the same problem with `y` that we did with `x` originally.

Resolving our dilemma

We have managed to show that `(plus (make-S y) (make-Z))` $\Rightarrow^*$ `(make-S (plus y (make-Z)))`.

We did this with a trace schema which gives a valid trace when we substitute any Nat `y`.

Suppose we had a trace showing `(plus y (make-Z))` $\Rightarrow^*$ `y` for a particular Nat `y`.

We can take this trace and wrap each expression in `(make-S ...)` to get a trace schema showing `(make-S (plus y (make-Z)))` $\Rightarrow^*$ `(make-S y)`.

Putting this trace together with the one we get from the trace schema above, we get a trace of `(plus (make-S y) (make-Z))` $\Rightarrow^*$ `(make-S y)`.

If we can show `(plus y (make-Z))` $\Rightarrow^*$ `y`,
then we can show `(plus (make-S y) (make-Z))` $\Rightarrow^*$ `(make-S y)`.

Proving the result we want

We want to show that for all Nats `x`, `(plus x (make-Z))` $\Rightarrow^*$ `x`.

We can show this if `x` is `(make-Z)`, by a trace.

We can then use the technique on the previous slide to show it if `x` is `(make-S (make-Z))`, and then `(make-S (make-S (make-Z)))`, and so on.

Since we can construct a trace for any given Nat, the result must be true for all Nats.

Again, our logical system has to consider this kind of reasoning valid.

Structural induction on Nats

Generalizing to other properties we may wish to prove, suppose we have a statement of the form “For all Nats `x`, P[`x`]”, where P is a description of a property of `x`.

For example, P[`x`] could be `(plus x (make-Z))` $\Rightarrow^*$ `x`.

If we can prove the following two things, we can conclude that the statement is true.

1. P[`(make-Z)`] is true, and
2. For all Nats `y`, if P[`y`] is true, then P[`(make-S y)`] is true.

Here P[`(make-Z)`] means P with `(make-Z)` substituted for `x`.

Another perspective: Nat is the smallest set containing `(make-Z)` and closed under `(make-S)`.

Other useful properties of Nats

With the technique of structural induction on Nats, we can prove many more things, such as the commutative and associative laws.

Commutative law:
For all Nats `x`, `y`, `z`, `(plus x y)` $\Rightarrow^*$ `z`
if and only if `(plus y x)` $\Rightarrow^*$ `z`.

Associative law:
For all Nats `x`, `y`, `z`, `w`, `(plus (plus x y) z)` $\Rightarrow^*$ `w`
if and only if `(plus x (plus y z))` $\Rightarrow^*$ `w`.

We take these properties for granted with natural numbers, but at some level, they do require justification.

Moving back to natural numbers

We can translate what we have learned with our simulation of natural numbers back to dealing with mathematical natural numbers and Racket's built-in numbers.

Data definition:
a natural number is either 0 or $m + 1$, where m is a natural number.

Template for a structurally-recursive function consuming a natural number:

```
(define (my-nat-fn n)
  (cond
    [(zero? n) ...]
    [(positive? n) ... (my-nat-fn (sub1 n)) ...]))
```

Mathematical induction:

To prove "For all natural numbers n, P[n]":

1. Prove P[0] is true, and
2. Prove for all natural numbers m, if P[m], then P[$m + 1$].

A function using structural recursion on natural numbers

Translating our `plus` function back to built-in numbers gives us a function that adds without using the built-in `+`.

```
(define (natural-plus n m)
  (cond
    [(zero? n) m]
    [(positive? n) (add1 (natural-plus (sub1 n) m))]))
```

This is not particularly useful, as it is less efficient than `+`. But it does demonstrate a useful pattern for computation with natural numbers.

Goals of this module

1. You should understand the syntax and semantics of structures and be able to write programs using them.
2. You should start to use templates in your programming.
3. You should be comfortable with recursive definitions and recursive functions, and start to use recursion in your programs as appropriate.
4. You should understand pure structural recursion based on a recursive data definition.
5. You should understand the use of for-all introduction and induction in proving things about recursive definitions and recursive functions.