Humans make their lives easier by organizing the things that they have to deal with into various categories.

We’ve seen a variety of these categories in Scheme.

Types of values in Scheme

- **Numbers**, e.g., 1, 3.14159
- **Symbols**, e.g., bill, hillary, al, tipper
- **Lists**, e.g., (bill hillary al tipper)
- **Booleans**, e.g., #t, #f
- **Strings**, e.g., "To be or not to be?"
- **Procedures**, e.g., #<procedure:+>
There are more things in Heaven and Earth…

There are more categories than the ones we’ve seen and, importantly, those categories can have rich structure.

A personnel database might need entries for name, birth_date, income, etc.

A graphics programming package might have data structures to represent a point, circle, rectangle, or line with entries for border_thickness, border_color, fill_pattern, etc.

Atomic data types

Many Scheme data types are atomic, i.e., they have no accessible internal structure:

- Numbers
- Booleans
- Symbols
- Procedures

Note that the lambda expression used to define a procedure might have a lot of structure, but the procedure that it evaluates to is atomic; we can’t look inside #<procedure:+>.

Non-atomic data types

The simplest non-atomic data structure in Scheme is the cons cell (or pair), which has two parts, the first and rest.

This is simple, but it’s enough to let use define lists in terms of cons cells – and we’ve done a lot with lists!

cons cells

Constructor: Built-in cons function.

Accessors: Built-in first and rest functions.

Type predicate: Built-in pair? function.
When we use cons cells to represent lists, we require the second argument of the cons function to be a list.

But recall that there's nothing illegal about using cons cells for other purposes, in which case the rest might be something else.

E.g.,
> (cons 1 2)
(1 . 2)
> (rest '(1 . 2))
2

Lists

Lists are a more complex data structure implemented in terms of cons cells.

Constructors:

- Built-in list function constructs a new list
- Built-in cons function pushes a new element onto the front of a list

Accessors: Primarily first and rest, but also lots of conveniences: second, third, ..., list-ref, list-tail, last,

Type predicate: Built-in list? function.

Beyond built-in data structures

Scheme allows programmers to define their own structures that can represent all kinds of objects that have a fixed number of properties.

We'll look at doing so in the context of a simple graphics application.
What is a rectangle?

It depends on what you want to do with it. Let’s suppose we want to draw it.

What information do we need?

- The *height* of the rectangle.
- The *width* of the rectangle.
- The *location* of the rectangle.
Using lists to implement the rectangle data type

> (load "graphics.scm")

;; Define a rectangle using the list
;; (left bottom width height)
> (define rect (list 10 10 20 40))
> (display-rectangle rect)
#t

[It displays in another window.]

What have we gained?

We’ve “glued” together several pieces of data to create a composite data object.

We’ve given the data a name, so we can refer to the composite data object as a whole and re-use it without specifying its contents each time.

```
(define display-rectangle
  (lambda (r)
    ;; Go to the position (x, y) for the rectangle’s bottom left corner.
    (goto (first r) (second r))

    ;; Draw the line from the current location in the bottom left corner
    ;; to the position (x, y) for the top left.
    (drawto (first r)
              (+ (second r) (fourth r))))

    ;; Draw a line from the current location in the top left corner
    ;; to the top right corner.
    (drawto (+ (first r) (third r))
            (+ (second r) (fourth r))))

    ;; Draw a line from the current location in the top right corner
    ;; to the bottom right corner.
    (drawto (+ (first r) (third r))
            (second r))

    ;; Draw a line from the current location in the bottom right corner
    ;; to the bottom left corner.
    (drawto (first r) (second r))))

That's a lot of firsts, seconds, etc.
```
Problems with this implementation

What if the programmer forgets the order of the data on the list? (left bottom width height)

What if the programmer forgets whether the data in the variable rect represents a rectangle or something else?

Rectangle data type, version 1

The rectangle data type

A rectangle is a structure with four fields, called “left”, “bottom”, “width”, and “height”. Together they define the size, shape, and position of the rectangle.

A structure definition is a new form of definition.

To define a rectangle as having the fields “left”, “bottom”, “width”, and “height”, we can use the following define-struct expression:

```
(define-struct rectangle (left bottom width height))
```
When a `define-struct` expression is evaluated, a new structure type is created with distinct constructor, predicate, accessor properties.

```
(define-struct rectangle (left bottom width height))
```

**Constructor:** `make-rectangle`

**Accessors:** `rectangle-left`, `rectangle-bottom`, `rectangle-width`, and `rectangle-height`

**Type predicate:** `rectangle?`

---

**Mutators: a note for later**

Evaluating a `define-struct` also creates *mutator* procedures for each field.

In this case, it creates `set-rectangle-left!`, `set-rectangle-bottom!`, `set-rectangle-width!`, and `set-rectangle-height!`

These are used for *destructive programming*, which we’ll discuss later.

---

**Defining the rectangle data type**

We can define a new data structure with “left”, “bottom”, “width”, and “height” fields

```
(define-struct rectangle (left bottom width height))
```

and Scheme automatically defines these procedures:

- `define make-rectangle (lambda (l b w h) ...)`
- `define rectangle-left (lambda (rectangle) ...)`
- `define rectangle-bottom (lambda (rectangle) ...)`
- `define rectangle-width (lambda (rectangle) ...)`
- `define rectangle-height (lambda (rectangle) ...)`
- `define rectangle? (lambda (thing) ...)`

---

**Rectangle structure example**

```
> (load "graphics.scm")
> (define rect (make-rectangle 10 10 20 40))
> rect
#<struct:rectangle>
> (display-rectangle rect)
#t
```

*[It displays in another window.]*
To be continued after Exam 2

Acknowledgments

This lecture incorporates material from:

Tom Ellman
Luke Hunsberger