Representing rectangles

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.

Rectangle

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td></td>
</tr>
<tr>
<td>bottom</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td></td>
</tr>
<tr>
<td>height</td>
<td></td>
</tr>
</tbody>
</table>
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>

[It displays in another window.]

More selector functions

;; The x coordinate for the right side is the
;; y coordinate for the left side plus the width
(define rectangle-right
  (lambda (r)
    (+ (rectangle-left r)
      (rectangle-width r))))

;; The y coordinate for the top is the y coordinate
;; for the bottom plus the height
(define rectangle-top
  (lambda (r)
    (+ (rectangle-bottom r)
      (rectangle-height r))))

Since right and top are not stored explicitly in the rectangle structure, they need to be computed “on the fly” from the explicitly stored values.
Rectangle data type, version 2

A different implementation of the rectangle data type

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

Defining the rectangle data type

We can define a new data structure with “left”, “bottom”, “right”, and “top” fields

(define-struct rectangle (left bottom right top))

and Scheme automatically defines these procedures:

(define make-rectangle (lambda (l b r t) ...))
(define rectangle-left (lambda (rectangle) ...))
(define rectangle-bottom (lambda (rectangle) ...))
(define rectangle-right (lambda (rectangle) ...))
(define rectangle-top (lambda (rectangle) ...))
(define rectangle? (lambda (thing) ...))
Rectangle structure example

> (define rect
  (make-rectangle 10 10 30 50))
> rect
#<struct:rectangle>
> (display-rectangle rect)
#t

[It displays in another window.]

More selector functions

(define rectangle-width
  (lambda (r)
    (- (rectangle-right r)
        (rectangle-left r))))

(define rectangle-height
  (lambda (r)
    (- (rectangle-top r)
        (rectangle-bottom r))))

Since width and height are not stored explicitly in the rectangle structure, they need to be computed “on the fly” from the explicitly stored values.

A procedure to draw a rectangle

(define display-rectangle
  (lambda (r)
    (goto (rectangle-left r) (rectangle-bottom r))
    (drawto (rectangle-left r) (rectangle-top r))
    (drawto (rectangle-right r) (rectangle-top r))
    (drawto (rectangle-right r) (rectangle-bottom r))
    (drawto (rectangle-left r) (rectangle-bottom r)))))

Notice that the rectangle drawing procedure doesn't need to change at all.

A common interface for rectangles
Comparing the two rectangle implementations

The first implementation stores \textit{left}, \textit{bottom}, \textit{width}, and \textit{height} in the structure and computes \textit{right} and \textit{top} on the fly.

The second implementation stores \textit{left}, \textit{bottom}, \textit{right}, and \textit{top} in the structure and computes \textit{width} and \textit{height} on the fly.

Selector functions have the same behavior in the two implementations.

The \texttt{make-rectangle} procedures are different in the two implementations.

Defining a uniform interface for constructing rectangles

\begin{verbatim}
(define build-rectangle
  (lambda (left bottom width height)
    (make-rectangle ...?...)))
\end{verbatim}

\texttt{build-rectangle} is called a \textbf{constructor}.

The constructor takes arguments \textit{l}, \textit{b}, \textit{w}, and \textit{h}.

We make no commitment regarding which of the quantities \textit{(l, b, w, h, r, t)} are stored explicitly and which are computed on the fly.

Constructor for the first implementation

\begin{verbatim}
(define-struct rectangle
  (left bottom width height))
(define build-rectangle
  (lambda (left bottom width height)
    (make-rectangle left bottom width height)))
\end{verbatim}

Constructor for the second implementation

\begin{verbatim}
(define-struct rectangle
  (left bottom right top))
(define build-rectangle
  (lambda (left bottom width height)
    (make-rectangle left bottom (+ left width)
      (+ bottom height))))
\end{verbatim}
Behavioral equivalence of the two implementations

We have two different implementations of the functions in the public interface.

- The function names are the same.
- The inputs are the same.
- The return values are the same.

The two implementations have the same behavior, even though the function definitions are different.

Rectangle data abstraction

The constructor and selector functions let us treat a rectangle as a **black box**. The data is inside the box. We access the data through the interface provided on the outside of the box. We can forget about what is actually inside the box.

**build-rectangle vs make-rectangle**

The *make-rectangle* procedure does *not* appear as a button on the black box; we can consider *make-rectangle* a **private constructor**.

The *build-rectangle* procedure *does* appear as a button on the black box; *build-rectangle* is a **public constructor**.

Representing triangles
What is a triangle?

Let's restrict ourselves to isosceles triangles, i.e., triangles with two equal length sides.

Let’s suppose we want to draw it.

What information do we need?
- Location of the base midpoint
- Width of the base
- Height of the base

The triangle data type

A triangle is a structure with four fields, called “apex”, “bottom”, “width”, and “height”. Together they define the size, shape, and position of the triangle.
Defining the triangle data type

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

(define-struct triangle (apex bottom width height))

and Scheme automatically defines these procedures:

(define make-triangle          (lambda (a b w h) ...))
(define triangle-apex           (lambda (triangle) ...))
(define triangle-bottom          (lambda (triangle) ...))
(define triangle-width           (lambda (triangle) ...))
(define triangle-height          (lambda (triangle) ...))
(define triangle?                (lambda (thing) ...))

Triangle structure example

> (define tri  
  (make-triangle 20 10 20 40))  
> tri  
#<struct:triangle>  
> (display-triangle tri)  
#t  
[It displays in another window.]

More selector functions

(define triangle-left           (lambda (t)  
  (- (triangle-apex t)  
    (/ (triangle-width t) 2))))
(define triangle-right           (lambda (t)  
  (+ (triangle-apex t)  
    (/ (triangle-width t) 2))))
(define triangle-top             (lambda (t)  
  (+ (triangle-bottom t) (triangle-height t))))

Since left, right, and top are not stored explicitly in the triangle structure, they need to be computed “on the fly” from the explicitly stored values.

A procedure to draw a triangle

(define display-triangle        (lambda (t)  
  (goto (triangle-left t)  
        (triangle-bottom t))  
  (drawto (triangle-apex t)  
          (triangle-top t))  
  (drawto (triangle-right t)  
          (triangle-bottom t))  
  (drawto (triangle-left t)  
          (triangle-bottom t))))

Notice that the programmer doesn’t need to think about how the structure is represented inside the computer. She only needs to remember the names of the selector functions.
The **triangle** data type

A triangle is a structure with four *fields*, called “left”, “bottom”, “right”, and “top”.
Together they define the size, shape, and position of the triangle.

---

**Defining the triangle data type**

We can define a new data structure with “left”, “bottom”, “right”, and “top” fields

```
(define-struct triangle (left bottom right top))
```

and Scheme automatically defines these procedures:

```
(define make-triangle (lambda (l b r t) ...))
(define triangle-left (lambda (triangle) ...))
(define triangle-bottom (lambda (triangle) ...))
(define triangle-right (lambda (triangle) ...))
(define triangle-top (lambda (triangle) ...))
(define triangle? (lambda (thing) ...))
```

---

**More selector functions**

```
(define triangle-apex
 (lambda (t)
   (/ (+ (triangle-left t) (triangle-right t)) 2))))
```

```
(define triangle-width
 (lambda (t)
   (- (triangle-right t) (triangle-left t))))
```

```
(define triangle-height
 (lambda (t)
   (- (triangle-top t) (triangle-bottom t))))
```

Since *apex*, *width*, and *height* are not stored explicitly in the triangle structure, they need to be computed “on the fly” from the explicitly stored values.
A procedure to draw a triangle

```
(define display-triangle
  (lambda (t)
    (goto (triangle-left t) (triangle-bottom t))
    (drawto (triangle-apex t) (triangle-top t))
    (drawto (triangle-right t) (triangle-bottom t))
    (drawto (triangle-left t) (triangle-bottom t))))
```

Notice that the triangle drawing procedure doesn't need to change at all.

Comparing the two triangle implementations

First implementation stores `apex`, `bottom`, `width`, and `height` in the structure and computes `left`, `right`, and `top` on the fly.

Second implementation stores `left`, `bottom`, `right`, and `top` in the structure and computes `apex`, `width`, and `height` on the fly.

Selector functions have the same behavior in the two implementations.

The `make-triangle` procedures are different in the two implementations.

Defining a uniform interface for constructing triangles

```
(define build-triangle
  (lambda (apex bottom width height) (make-triangle ...?...)))
```

`build-triangle` is called a constructor.

The constructor takes arguments `a`, `b`, `w`, and `h`.

We make no commitment regarding which of the quantities `(a, l, b, w, h, r, t)` are stored explicitly, and which are computed on the fly.
Constructor for the first implementation

```
(define-struct triangle
    (apex bottom width height))

(define build-triangle
    (lambda (apex bottom width height)
        (make-triangle apex bottom width height)))
```

Triangle apex bottom width height

---

Constructor for the second implementation

```
(define-struct triangle
    (left bottom right top))

(define build-triangle
    (lambda (apex bottom width height)
        (make-triangle
            (- apex (/ width 2)) bottom
            (+ apex (/ width 2)) (
                + bottom height))))
```

Triangle left bottom right top

---

Behavioral equivalence of the two implementations

We have two different implementations of the functions in the public interface.

The function names are the same.
The inputs are the same.
The return values are the same.

The two implementations have the same behavior, even though the function definitions are different.

---

Triangle data abstraction

The constructor and selector functions let us treat a triangle as a **black box**. The data is inside the box. We access the data through the interface provided on the outside of the box. We can forget about what is actually inside the box.
**build-triangle vs make-triangle**

As with our interface for rectangles:

The `make-triangle` procedure *does not* appear as a button on the black box; we can consider `make-triangle` a *private constructor*.

The `build-triangle` procedure *does* appear as a button on the black box; `build-triangle` is a *public constructor*.

---

**Public interface vs private implementation**

The procedures that appear as button on the black box constitute the *public interface* for the data type.

- People who use the rectangle and triangle packages are encouraged to write programs calling these procedures.

The procedures that don’t appear as buttons on the black box constitute the *private implementation* of the data type.

- People who use the rectangle and triangle packages should not write programs calling these procedures (though Scheme won’t prevent it).

---

**A parallel to functions**

We’ve written wrapper functions that call helper functions.

*The wrapper is the public interface.*

- It should be called directly or by other functions.

*The helper function is the private implementation.*

- Other functions shouldn’t call the helper.
  Sometimes we force the helper to be private by only defining it *inside* the wrapper using `letrec`. 
Data encapsulation

The public interface is said to \textit{encapsulate} the data in a “black box”.

It hides the private implementation from the programmer who uses the public interface.

Why is data encapsulation good?

Suppose I write triangle and rectangle constructor and selector functions using the first implementation of each data type.

Then I write a drawing program that constructs and draws rectangles and triangles.

Suppose I later change my mind and rewrite the constructor and selector functions using the second implementation of each type.

I don’t need to change my drawing program!

A procedure to draw objects

\begin{verbatim}
(define display-object
 (lambda (x)
  (cond ((rectangle? x)
           (display-rectangle x))
       ((triangle? x)
           (display-triangle x))
       (else #f))))
\end{verbatim}

Data abstraction

Building composite data structures out of simple ones.

Give the composite data structure a name.

Treat the composite data structure as a single object.

Write constructor and selector functions that hide the representation of the data structure.

Treat the object as a black box: We use the selector functions to access the data and forget about what’s actually inside the box.
Putting it together: Making houses

What is a house?
It depends on what you want to do with it.

Let's suppose we want to draw it.

Draw a rectangle.
Put an triangle on top of it.

What information do we need?
The location, width, and height of the rectangle.

We can compute the location, width, and height of the triangle from the description of the rectangle.

A minimalist house

Welcome to the neighborhood!

The house data type

A house is a structure with two fields, called “body” and “roof”.
The body is a rectangle.
The roof is a triangle.
**Defining the house data type**

Define a new data structure with “body” and “roof” fields:

```scheme
(define-struct house (body roof))
```

Scheme automatically defines the following procedures:

```scheme
(define make-house (lambda (body roof) ...))
(define house-body (lambda (house) ...))
(define house-roof (lambda (house) ...))
(define house? (lambda (thing) ...))
```

**Public constructor function**

Scheme defined `make-house`, but it doesn’t know how to make the appropriate triangle and rectangle!

```scheme
(define build-house (lambda (left bottom width height) (make-house (build-rectangle left bottom height width) (build-triangle (+ left (/ width 2)) (+ bottom height) width height))))
```

More amateur architecture on Wednesday!
Acknowledgments

This lecture incorporates material from:

- Tom Ellman
- Luke Hunsberger