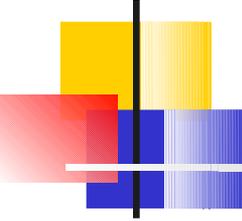


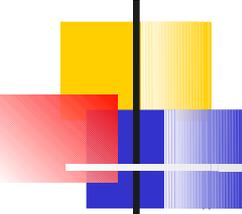
Data Definitions and Templates

Prof. Robert “Corky” Cartwright
Dr. Stephen Wong
Rice University



Recap of Previous Lecture

- Primitive types and values
 - numbers, booleans, symbols
- Variable definitions (constants), function definitions
- Operators
 - Arithmetic, relational, function application
- Rules for reducing programs
 - Leftmost reduction
- Conditional Expressions
- Syntax Errors & Runtime Errors



Challenge Problem from Previous Lecture

Can you think of a Scheme program that exhibits different behaviors with rightmost reduction instead of leftmost?

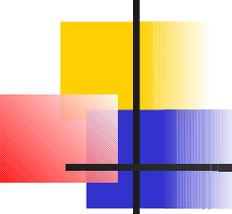
Consider the following example:

```
(+ (/ 1 0) (+ 'A 12))
```

Error conditions can make reasoning about programs more difficult than in naive mathematics e.g., you may not preserve program behavior by replacing `(* 0 (f x))` by `0`. What if evaluating `(f x)` generates an error?

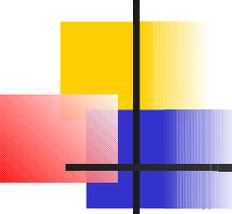
Another example: given a non-terminating function `omega`:

```
(+ (/ 1 0) (omega 0))
```



Goals of this lecture

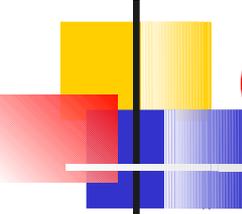
- Defining compound data (Scheme *structs*)
- Template for processing *structs*
- Union (mixed) data definitions
- Conditionals
- Template for processing union data.
- Inductive (self-referential) data definitions
- Template for processing inductive data



Simple Data Definitions

- How do we define new forms of data in Scheme? For example, say we want to write a program for the registrar that maintains a directory of courses that can be searched ...
- Informal rose description
“A complex number is a pair with a real part and an imaginary part , which are both numbers ”
- Corresponding data definition in Scheme

```
;; Complex is a structure (make-Complex real imag)
;; where real and imag are numbers
;; NOTE: the type complex is primitive in Scheme so we
;;      capitalize the name to avoid syntax errors
(define-struct Complex (real imag))
```
- A Scheme *struct* is a tuple tagged with the struct name
- Scheme processes this definition by creating the following operations:
 - *constructor*: `make-Complex`,
 - *accessors*: `Complex-real`, `Complex-imag`
 - *recognizer*: `Complex?` (which checks the tag)



Structs Can Represent Compound Data

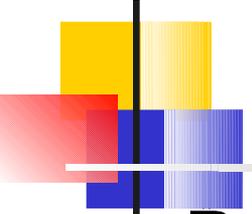
In the struct definition

```
(define-struct Complex (real imag))
```

real and **imag** are called **fields**.

If a struct has more than one field, it is a compound form of data because it more than one internal part. A struct with k fields can be thought of as a box with k compartments where each compartment is labeled with a distinct field name.

For example, the struct **Complex** has two fields (compartments) called **real** and **imag**.



Operations on Structures

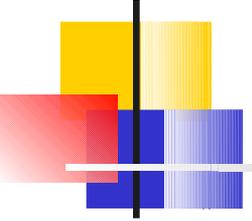
Recall that the following operations are automatically generated from the **define-struct** declaration for **Complex**

- *constructor*: **make-Complex**
- *accessors*: **Complex-real**, **Complex-imag**
- *recognizer*: **Complex?**

Sample reductions for these field accessors and structure recognizers

```
(Complex-imag (make-Complex 1 2)) => 2
```

```
(Complex? (make-Complex 3 4)) => true
```



Structures Are Values

- In a program, the structure returned by a constructor is a value and its parts are values.
- Inside a structure, the parts *must* be values. The application of a struct constructor like `make-Complex` to some argument expressions evaluates these arguments to produce *values*. At this point, the struct application becomes a *value* because all of the parts in its compartments are values.

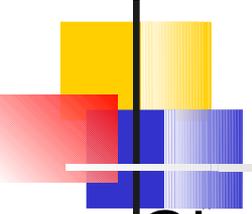
- For example:

`(make-Complex 0 (+ 2 2))` is a constructor application--not a value because the argument expression `(+ 2 2)` is not a value.

`(make-Complex 0 (+ 2 2)) => (make-Complex 0 4)`

`(make-Complex 0 4)` is a *value*. Why?

`(make-Complex x y)` is *not* a value. Why?



Evaluation Rules for Structures

Given the data definition

```
(define-struct Complex (real imag))
```

Scheme supports the following reduction rules:

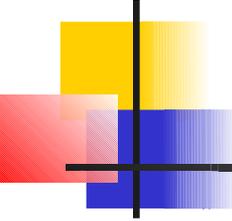
```
(Complex-real (make-Complex Val1 Val2)) => Val1
```

```
(Complex-imag (make-Complex Val1 Val2)) => Val2
```

```
(Complex? (make-Complex Val1 Val2)) => true
```

```
(Complex? Val3) => false
```

where `va11` `va12` are Scheme values and `va13` is not of the form `(make-Complex v1 v2)`

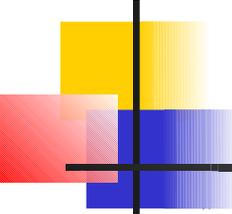


The Design Recipe

How should I go about writing programs?

- Analyze problem and **define** any requisite **data** types
- State the **type contract** and **purpose** for *function* that solves the problem
- Give **examples** of function use and result
- Select and **instantiate** a **template** for the function body
- Write the **code** for the function.
- **Test** the code, and confirm that tests succeeded

The order of the steps of the recipe is important. In DrScheme, steps 3 and 6 can be collapsed because the examples can be presented as calls on **check-expect**. DrScheme does not evaluate these tests until the end of the program text.



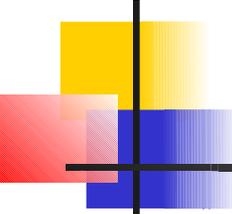
Template for Defined Data Type

- We start from the data definition. Example:

```
;; A Complex is a structure (make-Complex real imag)  
;; where real and imag are numbers  
(define-struct Complex (real imag))
```
- General template for any function processing an argument of type **Complex**

```
;; (define (f c)  
;;   ... (Complex-real c) ...  
;;   ... (Complex-imag c) ...)
```
- Type contracts for some possible functions on **Complex**

```
;; mag : Complex -> number  
;; 0? : Complex -> bool  
;; conj: Complex -> Complex
```



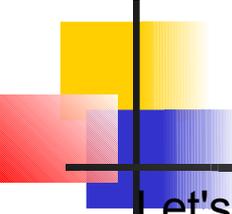
Example: write conj function

Assume that we have already defined the **Complex** type include a template for functions that process inputs of type **Complex**.

```
;; Type contract
;; conj: Complex -> Complex

;; Purpose: (conj c) conjugates the complex number c, i.e.,
;; (conj (make-Complex a b)) returns (make-Complex a (- b))

;; Examples:
(check-expect (conj (make-Complex 0 0)) (make-Complex 0 0))
(check-expect (conj (make-Complex 0 1)) (make-Complex 0 -1))
(check-expect (conj (make-Complex 0 -1)) (make-Complex 0 1))
(check-expect (conj (make-Complex 1 -1)) (make-Complex 1 1))
(check-expect (conj (make-Complex -1 1)) (make-Complex -1 -1))
```



Data Type → Template → Template Instantiation → Code

Let's follow the recipe for writing `conj`

```
;; Type contract:  
;; ... <as before>
```

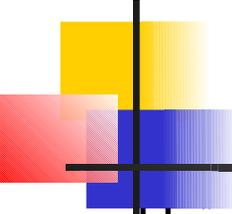
Instantiation of Complex template for `conj`

```
;; Template instantiation  
;; (define (conj c)  
;;   ... (Complex-real c) ...  
;;   ... (Complex-imag c) ...)
```

This template instantiation is trivial but more complex examples are not. It helps us write the code

```
;; Code:  
(define (conj c)  
  (make-Complex (Complex-real c) (- (Complex-imag c))))
```

- Sophisticated types → sophisticated templates ...
helping us write correct, sophisticated code



Union (Mixed) Data Definitions

How can we define data types that include more than one kind of data?

- Use the notion of *disjoint* set union from mathematics

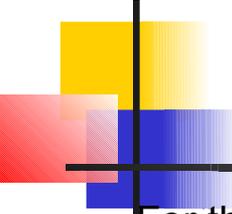
- Example:

```
;; A shape is either:  
;;   a square (make-square s) with side s,  
;;   an equilateral triangle (make-triangle s) with  
;;     side s, or  
;;   a circle (make-circle s) with diameter s,  
;; where s is a number and square, triangle, and circle  
;; are structs defined as follows.
```

```
(define square (size))  
(define triangle (size))  
(define circle (size))
```

- This data definition can be abbreviated as follows:

```
;; shape ::= (make-square s) | (make-triangle s) |  
;;         (make-circle s)  
;; where s is a number and square triange, and circle ...
```



Template for Union (Mixed) Data

For the type defined on the previous slide, the general template is:

```
; (define (f ... ashape ...)
; (cond
; [(square? ashape) ... (square-size ashape) ...] ;; square case
; [(triangle? ashape) ... (triangle-size ashape) ...] ;; triangle case
; [(circle? ashape) ... (circle-size ashape) ...])) ;; circle case
```

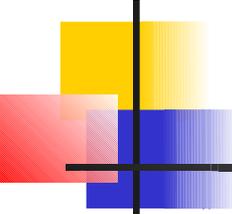
Processing mixed data requires a conditional to direct control to the appropriate case code. Note that **cond** is critical because it directs evaluation to the appropriate code, ignoring irrelevant clauses.

The template for an arbitrary union (assuming each construction is unary)

```
;; mixed-type ::= (make-S1 field) | ... | (make-SN field)
```

and struct definitions for S_1, \dots, S_N , the general template for processing data of this type is:

```
; (define (f ... amt ...)
; (cond
; [(S1? amt) ... (S1-field amt) ... ] ;; S1 case
; ... ;; cases 2, ..., N-1
; [(SN? amt) ... (SN-field amt) ... ])) ;; SN case)
```



Inductive Data Definitions

How can we generate arbitrarily large data objects like lists?

- Use self-reference (induction/recursion) in typing fields of union types. Such types are **inductive/recursive** types.

- Example:

```
:: A list-of-numbers is either
```

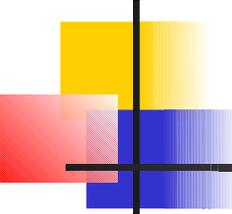
```
:: empty, or
```

```
:: (cons n lon)
```

```
:: where n is a number and lon is a list-of-numbers
```

- If we assume that **empty** is a built-in (primitive) constant (like **true**), this definition can be implemented in Scheme by the *struct*
(define-struct cons (first rest))

where **make-** is elided from the constructor and **cons-** is elided from the accessors.

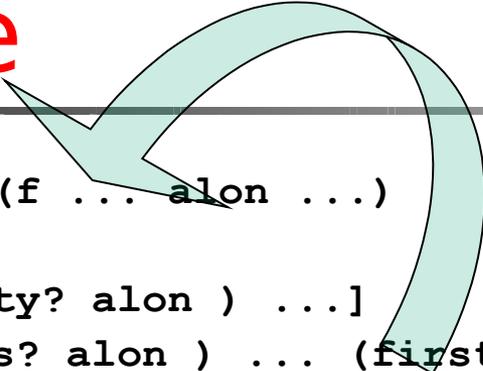
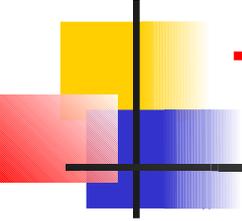


Inductive Data Definitions

The `cons struct` definition is built-in to Scheme; it is *primitive*.

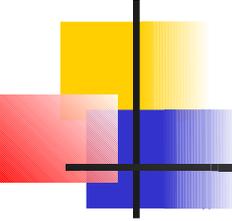
For the sake of brevity, the constructor is simply called `cons` rather than `make-cons` and the accessors are called `first` and `rest` rather than `cons-first` and `cons-rest`. Note that a Scheme *struct* definition does not stipulate the types of the fields of the structure. Hence, the programmer is responsible for ensuring that `cons` is used correctly. Moreover, a program can use `cons` in multiple ways. In our dialects of Scheme, `cons` ensures that its second argument is a list. (In the standard dialect, `first` is called `car` and `rest` is called `cdr` for historical reasons.)

Template for Inductive Data Type



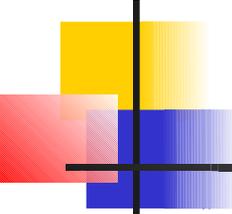
```
;; (define (f ... alon ...)
;; (cond
;; [(empty? alon) ...] ;; empty case
;; [(cons? alon) ... (first alon) ... ;; cons case
;; ... (f ... (rest alon) ...) ...]))
```

- Processing inductive (self-referential) data requires recursion (self-reference) in the computation.
- Recall the meaning of `cond`.
- This template for processing inductive data is an extension of the one on Slide 8 for processing a degenerate form of this template where there are multiple clauses (varieties) but no self-reference. The template is identical except for absence of the recursive call.



Extended Example: Insertion Sort

- Problem: given a list-of-numbers, sort it into ascending (non-decreasing) order.
- The solution that we will develop is the sample solution in the Scheme HW Guide.



If Expressions

- Simplified notation for common conditional expressions.

- Form:

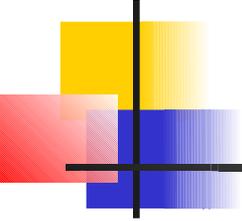
```
(if <question> <result-1> <result-2>)
```

abbreviates:

```
(cond [<question> <result-1>]  
      [else <result-2>])
```

Hence,

```
(if true <result-1> <result-2>) => <result-1>  
(if false <result-1> <result-2>) => <result-2>
```



Evaluation Rules for `if`

Rules for evaluating `if` expressions:

`(if true <expr1> <expr2>)` \Rightarrow `<expr1>`

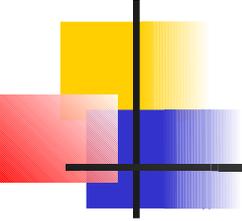
`(if false <expr1> <expr2>)` \Rightarrow `<expr2>`

`(if v <expr1> <expr2>)` \Rightarrow

error: question is not true or false

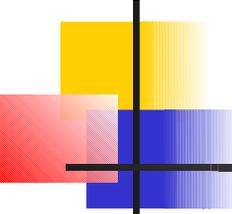
where `v` is a non-boolean value

Alternatively, we could expand an `if` expression into the equivalent `cond` expression, but this approach is clumsy.



Epilog

- Reminder: work on HW01. Over the weekend, you should be able to complete the problems from Section 8.3 and make substantial progress on the other programs. They all process lists.
- Next class: data-directed design using other inductive types



Example of a help function

```
; Type contract
; c-magnitude: Complex -> number
; Purpose: (c-magnitude c) computes the magnitude of the Complex number c, i.e.
; the L2 norm of (x,y) where c = (make-Complex x y)
; Examples
(check-expect (c-magnitude (make-Complex 0 0)) 0)
(check-expect (c-magnitude (make-Complex 3 4)) 5)
(check-expect (c-magnitude (make-Complex 4 3)) 5)
; Template Instantiation
; (define (c-magnitude c) ... (Complex-real c) ... (Complex-imag c) ...)
; Code
(define (c-magnitude c) (2norm (Complex-real c) (Complex-imag c)))
; Type contract
; 2norm: number number -> number
; Purpose: (2norm x y) returns the L2 norm of the vector (x,y), i.e.
; (sqrt (+ (* x x) (* y y)))
; Examples:
(check-expect (2norm 0 0) 0)
(check-expect (2norm 3 4) 5)
(check-expect (2norm 4 3) 5)
; Template instantiation: trivial
; Code:
(define (2norm x y) (sqrt (+ (* x x) (* y y))))
```