Data definitions

Prof. Robert “Corky” Cartwright
Prof. Vivek Sarkar
Department of Computer Science
Rice University
Recap of Previous Lecture

- Primitive types and values
  - numbers, booleans, symbols
- Variable definitions, function definitions
- Operators
  - Arithmetic, relational, function application
- Rules for reducing programs
  - Leftmost reduction
- Syntax Errors & Runtime Errors
- Conditional Expressions
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 different from standard math e.g., you may not always preserve program behavior by replacing \((* 0 (f x))\) by 0

COMP 211, Spring 2010
Today’s Goals

• Compound data definitions and templates
  • Structures
• Inductive (self-referential) compound data definitions and templates
  • Lists
• if expressions
Compound Data: Structures

- Scheme structures can be used to combine a fixed number of values into a single piece of data e.g.,

- **Problem description**
  - “A complex number has a real part and an imaginary part”

- **Data definition**
  
  ```scheme
  ;; cmplx is a structure (make-cmplxx real imag)
  ;; where real and imag are real numbers
  (define-struct cmplx (real imag))
  ```

  The structure, cmplx, contains two numbers.
Operations on Structures

- The following operations are automatically generated from the define-struct declaration for cmplx:
  - constructor: make-cmplx
  - accessor: cmplx-real, cmplx-imag
  - recognizer: cmplx?

- Reductions for field accessors and structure recognizers:
  
  (cmplx-imag (make-cmplx 1 2)) => 2
  
  (cmplx? (make-cmplx 3 4)) => true
Structures are values

- A structure returned by a constructor is a value (and hence is \textit{not} reducible)
  - A structure is like a box with a value in each compartment
  - It may be big, but it’s just like 1, true, or ‘Rabbit
  - It may be big, but it is \textit{NOT} a reducible expression, like (+ 1 2)

- Notes:
  - (make-cmplx 1 2) is a value
  - (make-cmplx x y) is \textit{not} a value (why not?)
  - (make-cmplx 10 (+ 25 25)) is \textit{not} a value (why not?)
An Aside: Converting our cmplx structure to a complex number in DrScheme

;; define the cmplx structure
(define-struct cmplx (real imag))

;; define eval function for converting cmplx to a Scheme number
(define (eval z) (+ (cmplx-real z) (* (cmplx-imag z) (sqrt -1)))))

;; define variables C1, C2
(define C1 (make-cmplx 1 2))
(define C2 (make-cmplx 1 -2))

;; evaluate C1 and C1*C2
> (eval C1)
1+2i
> (* (eval C1) (eval C2))
5
Template for Defined Data Types

• We start from the data definition. Example:

;;; A course is a structure (make-course dept num size)
;;; where dept is a symbol, and num and size are numbers
(define-struct course (dept num size))

• A function template must include a model of all operations that can be performed on input structure arguments e.g., here’s a template for function \( f \) with argument \( c \) of type course

;;; (define (f c)
;;;     ... (course-dept c) ...
;;;     ... (course-num c) ...
;;;     ... (course-size c) ...)

COMP 211, Spring 2010 9
Type --&gt; Template --&gt; Code

- Template for function processing a course
  
  ```scheme
  ;; (define (f ... c ... )
  ;;     ... (course-dept c) ...
  ;;     ... (course-num c) ...
  ;;     ... (course-size c) ...)
  ```

- Instantiation of template for big-class?
  
  ```scheme
  ;; (define (big-class? c)
  ;;     ... (course-dept c) ...
  ;;     ... (course-num c) ...
  ;;     ... (course-size c) ...)
  ```

- Templates help us write the code
  
  ```scheme
  (define (big-class? c) (>= (course-size c) 30))
  ```

- Sophisticated types -> sophisticated templates ...
  helping us write correct, sophisticated code
Structures can be nested

```
(define-struct course (dept num size))
(define-struct department (school numFaculty numMajors))
```
Limitations of structures

• Structures cannot contain variable numbers of elements
• Structures are impractical for large numbers of elements e.g.,

```
(define-struct BeerBikeTeam
  (rider1 rider2 ... rider10
  chugger1 chugger2 ... chugger10
  ...
)
)
```
Lists: defining Compound Data with Variable Number of Elements

• How can we generate arbitrarily large data objects like lists?
• Use a two-element struct as a building block to chain together multiple elements e.g.,
  (define-struct cons (first rest))

```
(define-struct cons (first rest))
```

```
  first  rest
```

```
Biker1  first  rest
```

```
Biker2  first  rest
```

```
Biker10  first  rest
```

```
empty
```
Inductive Data Definitions for Lists

- Use self-reference (induction/recursion)
- Example:

```clojure
;; A list-of-numbers is either
;;   empty, or
;;   (cons n lon)
;; where n is a number and lon is a list-of-numbers
```
Built-in support for lists in Scheme

- `cons` is a built-in `struct` definition in Scheme, with special abbreviated names for its operations.

<table>
<thead>
<tr>
<th>Normal struct operation</th>
<th>Equivalent list operation (what you should use)</th>
</tr>
</thead>
<tbody>
<tr>
<td>make-cons</td>
<td><code>cons</code> Can also use <code>'(e1 e2 ...)</code> as shorthand for a list, instead of nested cons operations</td>
</tr>
<tr>
<td>cons-first</td>
<td><code>first</code></td>
</tr>
<tr>
<td>cons-rest</td>
<td><code>rest</code> (teaching dialects of Scheme also check that rest is a list)</td>
</tr>
<tr>
<td>cons?</td>
<td><code>cons?</code></td>
</tr>
</tbody>
</table>
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.
If Expressions

• Simplified notation for common conditional expressions.
• Form:
  \[(if \text{ question} \ \text{result-1} \ \text{result-2})\]

abbreviates:

\[(cond \ [\text{question} \ \text{result-1}] \ [\text{else} \ \text{result-2}])\]

• Hence,

\[(if \ \text{true} \ \text{result-1} \ \text{result-2}) \Rightarrow \ \text{result-1}\]
\[(if \ \text{false} \ \text{result-1} \ \text{result-2}) \Rightarrow \ \text{result-2}\]
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.
  - https://wiki.rice.edu/confluence/display/cswiki/211Guidelines
Auxiliary function: insert

;;; Contract and purpose
;;; insert: number list-of-numbers -> list-of-numbers
;;; Purpose: (insert n alon), where alon is sorted in ascending order, returns a list containing n and the elements of alon also sorted in ascending order

;;; Examples and Tests:
(check-expect (insert 17 empty) '(17))
(check-expect (insert 17 '(17)) '(17 17))
(check-expect (insert 4 '(1 2 3)) '(1 2 3 4))
(check-expect (insert 0 '(1 2 3)) '(0 1 2 3))
(check-expect (insert 2 '(1 1 3 4)) '(1 1 2 3 4))

COMP 211, Spring 2010
Auxiliary function: insert (contd)

;; Template instantiation
(define (insert n a-lon)
  (cond
    [(empty? a-lon) ...]
    [(cons? a-lon) ... (first a-lon) ...
      ... (insert n (rest a-lon)) ... ]))

;; Code
(define (insert n a-lon)
  (cond
    [(empty? a-lon) (cons n empty)]
    [(cons? a-lon)
      (if (<= n (first a-lon)) (cons n a-lon)
        (cons (first a-lon) (insert n (rest a-lon))))]))
Main function: sort

;; Main function: sort

;; Contract and purpose:
;; sort: list-of-numbers -> list-of-numbers
;; Purpose: (sort alon) returns the a list with same elements
;; (including duplicates) as alon but in ascending order.

;; Examples and Tests:
(check-expect (sort empty) empty)
(check-expect (sort '(0)) '(0))
(check-expect (sort '(1 2 3)) '(1 2 3))
(check-expect (sort '(3 2 1)) '(1 2 3))
(check-expect (sort '(10 -1 10 -20 5)) '(-20 -1 5 10 10))

COMP 211, Spring 2010
Main function: sort (contd)

(define (sort a-lon)
  (cond
    [(empty? a-lon) empty]
    [(cons? a-lon) (insert (first a-lon) (sort (rest a-lon)))]))
The Design Recipe (Again!)

How should I go about writing programs?

1. Analyze problem and define any requisite data types
2. State contract (type) and purpose for function that solves the problem
3. Give examples of function use and result
4. Select and instantiate a template for the function body
5. Write the function itself
6. Test it, and confirm that tests succeeded

The order of the steps of the recipe is important
Announcements

- Monday (Jan 18\textsuperscript{th}) is a holiday
  - No lecture on Monday
  - Monday labs have been rescheduled to Wednesday (Jan 20\textsuperscript{th})
- Reminder: work on HW01, due at 10am on Jan 22\textsuperscript{nd}
  - See course homework guidelines for details, especially hand evaluation problems
    - https://wiki.rice.edu/confluence/display/cswiki/211Guidelines
- Next class on Jan 20\textsuperscript{th}: data-directed design using other inductive types