Data definitions and Conditionals

Prof. Robert “Corky” Cartwright
Department of Computer Science
Rice University

Today's Goals

- Simple data definitions
- Template for processing simple data
- Inductive (self-referential) data definitions
- Conditionals
- 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 …
- Problem description
  - "… Each university course will have an associated department and course numbers, as well as a class size. …"
- Data definition

```
;; 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))
```

- Scheme processes this definition by creating the following operations:
  - *constructor:* `make-course`,
  - *accessors:* `course-dept`, `course-num`, `course-size`
  - *recognizer:* `course?`

Creating and Using Structures

- Syntax for creating a structure:

```
(define this-class (make-course 'COMP 211 41))
```

- A structure (a constructor applied to values) is a value (and hence is *not* reducible)
  - It's big. But it's just like `1, true, or 'Rabbit`
  - It's big. But it is NOT a reducible expression, like `(+ 1 2)`
- Syntax for extracting fields
  - `(course-dept this-class)`
    `(course-num this-class)`
- Reduction for field access

```
(course-dept (make-course 'COMP 210 50)) => 'COMP
```

- Notes:
  - `(make-course 'COMP 210 50)` is a value
  - `(make-course 'COMP 210 size)` is *not* a value (why not?)
  - `(make-course 'COMP 210 (+ 25 25))` is *not* a value (why not?)

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

Template for Defined Data Type

- 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))
```

- Template for <u>any</u> function <u>processing</u> an argument of type `course`

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

- Examples of such a function

```
;; big-class? : course -> bool
;; empty-class? : course -> bool
;; change-dept : course dept -> course
```

Type --> Template --> Code

- Template for function processing a `course`

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

- Instantiation of template for `big-class?`

```
;; (define (big-class? c)
;;     ... (course-dept c) ...
;;     ... (course-num c) ...
;;     ... (course-size c) ...)
```

- Templates help us write the code

```
(define (big-class? c) (>= (course-size c) 30))
```

- Sophisticated types -> sophisticated templates … helping us write correct, sophisticated code

Inductive Data Definitions

- How can we generate arbitrarily large data objects like lists?
- Use self-reference (induction/recursion)
- 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 constant (like `true`), this definition can be implemented in Scheme by the `struct`

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

- This `struct` definition is built-in to Scheme (a 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. In teaching dialects of Scheme, `cons` ensures that its second argument is a list.

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.
- What is `cond` ?
- There is a degenerate form of the data definition given on the previous slide where there are multiple clauses (varieties) but no self-reference. The template emplate is identical except for absence of the recursive call.

Conditional Expressions

- Mechanism for distinguishing different forms of input.
- Form:

```
(cond [question-1  result-1]
      [question-2  result-2]
      ...
      [question-n  result-n]
      [else        default-result])
```

- Square brackets are used above for clarity. In Scheme, they are synonymous with parentheses, but balancing brackes must match.
- `else` is optional. If omitted and none of the questions are true, the result is a run-time error (like division by zero).

Reduction of Conditional Expressions

```
(cond [true         result-1]
      [...           ...])
=>
result-1

(cond [false        result-1]
      [question-2   result-2]
      ...
      [else         default-result])
=>
(cond [question-2   result-2]
      ...
      [else         default-result])

(cond [false        result-1]
      [else         default-result])
=>
default-result
```

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
```

Extended Example: Insertion Sort

- Problem: given a list-of-numbers, sort it into ascending (non-decreasing) order.

Epilog

- Reminder: work on HW01. Over the weekend, you should be able to complete problems from Section 8.3 and make substantial progress on the programs. They all process lists.
- Next class: data-directed design using other inductive types
- Bonus class meeting
  Possible times: W 2, W 3,
  Tu 9, Tu 10, Tu 11, Tu 1, Tu 2, Tu 3