CSCI-C311 Programming Languages

Racket: struct

Dr. Hang Dinh

1

Reading Assignment for This Lecture

- The Racket Guide
  - https://docs.racket-lang.org/guide/index.html
  - Part 5. Programmer-Defined Datatypes

2

Simple Structure Type: struct

- New datatypes in Racket are normally created with the struct form
  - Even classes and objects are implemented in terms of structure types.
  - Structure type is a composite data structure that has a set of predefined *fields*.
- Syntax of struct

```
(struct struct-id (field-id ...))
```

  - *struct-id* is a *constructor* function that takes as many arguments as the number of *field-id*s, and returns an instance of the structure type.
- Example:

```
(struct point (x y))
```

```
> (point 1 2)
#<point>
```

3

Reflective Operations and Accessors of Structure Types

- Given a definition of struct

```
(struct struct-id (field-id ...))
```

  - *struct-id?* is a *predicate* function that takes a single argument and returns #t if it is an instance of the structure type, #f otherwise.
  - *struct-id-field-id*, for each *field-id*, is an *accessor* that extracts the value of the corresponding field from an instance of the structure type.

```
> (point? 1)
#f
> (point? (point 1 2))
#t
```

```
> (point-x (point 1 2))
1
> (point-y (point 1 2))
2
```

4

Copy and Update Structure Instances

- The `struct-copy` form clones a structure and optionally updates specified fields in the clone.

  `(struct-copy struct-id struct-expr [field-id expr] ...)`

  - *struct-id* must be a structure type name bound by `struct`
  - *struct-expr* must produce an instance of the structure type.
  - The result is a new instance of the structure type that is like the old one, except that each *field-id* gets the value of the corresponding *expr*.

```
(define p1 (point 1 2))
(define p2
  (struct-copy point p1 [x 3]))
```

```
> (point-x p2)
3
> (point-y p2)
2
```

5

Structure Subtypes (kind of inheritance)

- Structure subtype is a structure type that extends an existing one
  - A structure subtype inherits the fields of its supertype
- Use an extended form of `struct` to define a structure subtype

  `(struct struct-id super-id (field-id ...))`

  - *super-id* must be a structure type name bound by `struct`
- Example:

```
(struct point (x y))
(struct 3Dpoint point (z))
```

6

Structure Subtypes (kind of Inheritance)

- The subtype constructor accepts the values for the subtype fields after values for the supertype fields.

```
> (define p (3Dpoint 1 2 3))
> (3Dpoint-z p)
3
```

- An instance of a structure subtype can be used with the predicate and accessors of the supertype.

```
> (point? p)
#t
> (point-x p)
1
```

```
> (3Dpoint-x p)
3Dpoint-x: undefined;
cannot reference an identifier before its definition
```

7

Opaque vs. Transparent Structure Types

- Structure types by default are *opaque* (i.e., private)
  - Their instances prints won't show any information about the fields' values
  - Their accessors and mutators of a structure type are kept private to a module
- To make a structure type *transparent*, add the `#:transparent` keyword after the field-name sequence

```
(struct point (x y))
```

```
> (point 1 2)
#<point>
```

```
(struct point (x y)#:transparent)
```

```
> (point 1 2)
(point 1 2)
```

8

Transparent Structure Instances Comparisons

- For a transparent structure type, the `equal?` function returns true iff both instances have the same field values.

```
(struct point (x y)#:transparent)
(define p1 (point 1 2))
```

```
> (equal? p1 (point 1 2))
#t
> (equal? (point 1 2) (point 1 2))
#t
> (equal? (point 1 2) (point 1 3))
#f
```

9

Opaque Structure Instances Comparisons

- For opaque structure type, the `equal?` function returns true iff both instances have the same identify.

```
(struct date (month day year))
(define d1 (date "Feb" 10 2022))
```

```
> (equal? d1 d1)
#t
> (equal? d1 (date "Feb" 10 2022))
#f
> (equal? (date "Feb" 10 2022) (date "Feb" 10 2022))
#f
```

10

Mutable Fields

- By default, all fields in a structure type are *immutable*
- To make a field mutable, add keyword `#:mutable` to that field in the structure type definition

```
(struct date ([month #:mutable] [day #:mutable] year))
```

- To make all fields of the structure to be mutable, add `#:mutable` after the field-name sequence

```
(struct date (month day year) #:mutable)
(struct point (x y) #:mutable #transparent)
```

11

Mutators

- For each mutable field, Racket automatically generates a *mutator*:
  - Whose name is of the form `set-`*`struct-id-field-id`*!
  - It sets the value of *`field-id`* in an instance of the *`struct-id`* structure.

```
(struct date ([month #:mutable] [day #:mutable] year))
(define d1 (date "Feb" 10 2022))
```

```
> (date-month d1)
"Feb"
> (set-date-month! d1 "March")
> (date-month d1)
"March"
> (set-date-year! d1 2023)
set-date-year!: undefined;
 cannot reference an identifier before its definition
```

12

Automatic Fields

- An automatic field is indicated by the `#:auto` field option
  - Use `#:auto-value` option to specify a value to be used for ALL auto fields

```
(struct student (name id [age #:auto] [course #:auto])
                        #:auto-value 18)
```

```
> (define s1 (student "Robert" 101000))
> (student-age s1)
18
> (student-course s1)
18
```

The constructor procedure does not accept arguments for automatic fields.

```
> (struct student ([name #:auto] id [course #:auto]))
struct: non-auto field after an auto field disallowed in: id
```

- Automatic fields are implicitly mutable but don't have mutators.

13

Constructor Guard

- A constructor guard is a procedure to be called whenever an instance of the structure type is created.
  - It provides a mechanism to validate the field values
- A constructor guard is created with the keyword `#:guard`
  - The guard takes as many arguments as non-auto fields in the structure type, plus one more for the name of the instantiated type.
  - It must return the same number of values as given, minus the name argument

```
(struct date (month day [year #:auto])#:auto-value 2020
  #:guard (lambda (m d typename)
            (values m d)
            ))
```

Function `values` returns its provided arguments

14

Constructor Guard

- The guard can raise an exception if one of the given arguments is unacceptable, or it can convert an argument.

```
(struct date (month day [year #:auto])#:auto-value 2020
  #:transparent
  #:guard (lambda (m d typename)
            (cond [(< d 1) (values m 1)]
                  [(> d 31)
                   (error typename
                          "Day cannot be larger than 31: ~e" d)]
                  [else (values m d )])))
```

```
> (date "Feb" 0)
(date "Feb" 1 2020)
> (date "Feb" 32)
date: Day cannot be larger than 31: 32
```

15

Constructor Guard for Structure Subtypes

```
(struct position (x y)
  #:guard (lambda (x y name)
            (if (not (and (number? x) (number? y)))
                (error name "Both fields x and y must be numbers.")
                (values x y))))

(struct 3Dposition position (z) #:transparent
  #:guard (lambda (x y z name)
            (if (number? z)
                (values x y z)
                (if (string? z)
                    (values x y (string->number z))
                    (error name "Invalid value for y:~e" z)))))
```

The guard for the subtype must take arguments for the fields of the supertype.

```
> (3Dposition 1 2 "a")
(3Dposition ... #f)
> (3Dposition "a" 2 "c")
3Dposition: Both fields x and y must be numbers.
> (3Dposition 1 2 #f)
3Dposition: Invalid value for y:#f
```

Subtype's name is reported by the supertype's guard

16