CSc 520

Principles of Programming Languages

6: Scheme — Symbols and Structures

Christian Collberg
collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

Copyright © 2005 Christian Collberg
Symbols

- In addition to numbers, strings, and booleans, Scheme has a primitive data-type (*atom*) called *symbol*.

- A symbol is a lot like a string. It is written:

  `'identifier`

- Here are some examples:

  `'apple`
  `'pear`
  `'automobile`

- `(symbol? arg)` checks if an atom is a symbol.

- To compare two symbols for equality, use `(eq? arg1 arg2)`. HTDP says to use `(symbol=? arg1 arg2)` but DrScheme doesn’t seem to support this.
Symbols...

> (symbol? "hello")
#f

> (symbol? 'apple)
#t

> (eq? 'a 'a)
#t

> (eq? 'a 'b)
#f

> (display 'apple)
apple

> (string->symbol "apple")
apple

> (symbol->string 'apple)
"apple"
(define (healthy? f)
  (case f
    [(sushi sashimi) 'hell-yeah]
    [(coke) 'I-wish]
    [(licorice) 'no-but-yummy]
    [else 'nope]
  )))

> (healthy? 'sashimi)
hell-yeah
> (healthy? 'coke)
i-wish
> (healthy? 'licorice)
no-but-yummy
> (healthy? 'pepsi)
nope
Structures

Some versions of Scheme have **structures**. Select **Advanced Student** in DrScheme.

These are similar to C’s `struct`, and Java’s `class` (but without inheritance and methods).

Use `define-struct` to define a structure:

```
(define-struct struct-name (f1 f2 ...))
```

`define-struct` will automatically define a constructor:

```
(make-struct-name (f1 f2 ...))
```

and field-selectors:

```
struct-name-f1
struct-name-f2
```
(define-struct person (name sex date-of-birth))

> (define bob (make-person "bob" 'male '1978))
> bob
(make-person "bob" 'male '1978)
> (define alice (make-person "alice" 'female '1979))

> (person-sex bob)
'male
> (person-date-of-birth alice)
'1979
Equivalence

Every language definition has to struggle with equivalence; i.e. what does it mean for two language elements to be the same?

In Java, consider the following example:

```java
void M(String s1, String s2, int i1, int i2) {
    if (i1 == i2) . . . ;
    if (s1 == s2) . . . ;
    if (s1.equals(s2)) . . . ;
}
```

Why can I use == to compare ints, but it is it usually wrong to use it to compare strings?
Equivalence.

Scheme has three equivalence predicates `eq?`, `eqv?` and `equal?`.

`eq?` is the pickiest of the three, then comes `eqv?`, and last `equal?`.

In other words,

- If `(equal? a b)` returns `#t`, then so will `(eq? a b)` and `(eqv? a b)`.
- If `(eqv? a b)` returns `#t`, then so will `(eq? a b)`.

`(equal? a b)` generally returns `#t` if `a` and `b` are structurally the same, i.e. print the same.
Equivalence...

(eqv? a b) returns #t if:
- a and b are both #t or both #f.
- a and b are both symbols with the same name.
- a and b are both the same number.
- a and b are strings that denote the same locations in the store.

> (define S "hello")
> (eqv? S S)
true
> (eqv? "hello" "hello")
false
> (eqv? 'hello 'hello)
true
Equivalence...

- `(equal? a b)` returns `#t` if `a` and `b` are strings that print the same.
- This is known as **structural equivalence**.

> `(equal? "hello" "hello")`
true

> `(equal? alice bob)`
false

> `(define alice1 (make-person "alice" 'female '1979))`
> `(define alice2 (make-person "alice" 'female '1979))`
> `(equal? alice1 alice2)`
true