ECE570 Lecture 2: Types and Conventions

Jeffrey Mark Siskind

School of Electrical and Computer Engineering

Fall 2013

Booleans

SCHEME has procedure to operate on Boolean values:

`(and #t #f)` $\Longrightarrow$ `#f`

`(or #t #f)` $\Longrightarrow$ `#t`

`(not #t)` $\Longrightarrow$ `#f`

Note: `and` and `or` are not really procedures. More on that later.

Truth

- `#f` is the only SCHEME value treated as **false**.
- (In some implementations the empty list `()` is also treated as **false**.)
- All other SCHEME values are treated as **true**.
- Builtin predicates like =, <, >, <=, >=, etc. always yield `#t` or `#f`, never another true or false value.

`(if #t 1 2)` $\Longrightarrow$ `1`

`(if #f 1 2)` $\Longrightarrow$ `2`

`(if 27 1 2)` $\Longrightarrow$ `1`

Gotcha: In some implementations: `(if '() 1 2)` $\Longrightarrow$ `1`
In other implementations: `(if '() 1 2)` $\Longrightarrow$ `2`

AND and OR are Syntax

- `(and (> x 0) (< y 5))` is **syntax**.
- `(or (> x 0) (< y 5))` is **syntax**.
- `(not (> x 0))` is a procedure call.
- `and` and `or` do *short-circuit* evaluation.
  - `(and (> x 0) (< (sqrt x) 42))`
  - left-to-right
  - `and` stops at first false subexpression.
  - `or` stops at first true subexpression.
- `and` and `or` take zero or more arguments.
  - `(and)` $\Longrightarrow$?
  - `(or)` $\Longrightarrow$?

AND Can Return Something Other Than #T or #F

- `and` returns the value of the first false subexpression if there is one.
  - `(and (< 3 2) 5)` $\Longrightarrow$ `#f`
- `and` returns the value of the last subexpression if all subexpressions are true.
  - `(and (> 3 2) 5)` $\Longrightarrow$ `5`
  - `(and 5 (> 3 2))` $\Longrightarrow$ `#t`

Gotcha: `and` is implementation dependent when given the empty list `()` as an argument.

OR Can Return Something Other Than #T or #F

- `or` returns the value of the first true subexpression if there is one.
  - `(or 5 (> 3 2))` $\Longrightarrow$ `5`
  - `(or (> 3 2) 5)` $\Longrightarrow$ `#t`
- `or` returns the value of the last subexpression if all subexpressions are false.
  - `(or (< 7 6) (< 3 2))` $\Longrightarrow$ `#f`

Gotcha: `or` is implementation dependent when given the empty list `()` as an argument.

The Numeric Tower—I

SCHEME has arbitrary precision integers (bignums),

`(factorial 30)` $\Longrightarrow$ `265252859812191058636308480000000`

rationals,

`(/ 1 2)` $\Longrightarrow$ `1/2`

reals, and

`(sqrt 2)` $\Longrightarrow$ `1.4142135623730951`

complex numbers.

`(+ 1 (sqrt -4))` $\Longrightarrow$ `1+2i`

The numerator and denominator of a rational can be bignums.

The real and imaginary components of a complex number can be rational or real.

The Numeric Tower—II

Numeric type is a property of the number, not its representation.

`(integer? 2/1)` $\Longrightarrow$ `#t`

`(integer? 1.0)` $\Longrightarrow$ `#t`

`(integer? 1+0i)` $\Longrightarrow$ `#t`

`(rational? 1.0)` $\Longrightarrow$ `#t`

`(rational? (sqrt 2.0))` $\Longrightarrow$ `#t`

`(rational? (sqrt 2))` $\Longrightarrow$ `#t`

`(rational? 1+0i)` $\Longrightarrow$ `#t`

The Numeric Tower—III

All integers are rational, all rationals are real, all reals are complex, and all complex numbers are numbers.

```
(rational? 1) ⟹ #t

(real? 1) ⟹ #t

(real? 1/2) ⟹ #t

(complex? 1) ⟹ #t

(complex? 1/2) ⟹ #t

(complex? (sqrt 2)) ⟹ #t
```

The Numeric Tower—IV

SCHEME supports exponential notation:

```
(expt 10.0 -20) ⟹ 1e-20
```

The Numeric Tower—V

SCHEME numbers are *exact* or *inexact*.

(exact? 1) $\Longrightarrow$ #t

(inexact? 1.0) $\Longrightarrow$ #t

(inexact? 1e5) $\Longrightarrow$ #t

(exact? 1/3) $\Longrightarrow$ #t

(exact? 1+1/2i) $\Longrightarrow$ #t

(inexact? 1.0+1/2i) $\Longrightarrow$ #t

The Numeric Tower—VI

Exactness is orthogonal to numeric type.

Implementations are not required to support all numeric types and all combinations of numeric type and exactness.

Implementations can silently coerce exact numbers to inexact numbers but must preserve inexactness.

Inexactness is lost when explicitly leaving the number domain or by using the `inexact->exact` type coercion.

The precise rules are complicated. See the manual for details.

We will only use exact integers and inexact rationals in this course.

Types

`1` is a *number*.

(Actually, it is a numeric *constant* that evaluates to a number.)

+ is a *procedure* that takes numeric arguments and results a numeric result.

(Actually, it is a *variable* that is bound to such a procedure.)

< is a procedure that returns a *Boolean* result.

`#t` and `#f` are Boolean constants.

(Non-Boolean values are often treated as true. There are exceptions, however. The precise rules are complicated. See the manual for details.)

Number and Boolean are *types*.

Conventions—I

Convention: Procedures that return Boolean values have names that end in '`?`'.

Examples:	`number?`, `exact?`, `integer?`, `boolean?`
exceptions:	=, <, <=, >, >=

Conventions are not enforced but it is a good idea to follow them. It helps other people understand your code. As a newcomer, it is a good idea to to follow the established conventions of a community because it helps you become a member of that community. It also allows you to leverage the existing wisdom and experience of that community.

Conventions—II

Convention: Variables that are bound to Boolean values have names that end in '`?`'.

Holds for global variables,

Examples: `(define *lifted?* #f)`

local variables,

Examples: `(let ((entry? #f)) ...)`

procedure parameters, and

Examples: `(define (tracking-pointer twice? press? tracker) ...)`

structure slots.

Examples: `(define-structure pbm raw? bitmap)`

Conventions—III

Convention: Variables that are bound to procedures that return Boolean values have names that end in '?'.

Because:

```
(define (bound? x)
 (null? (rest (domain-variable-domain x))))
```

is really shorthand for:

```
(define bound?
 (lambda (x)
  (null? (rest (domain-variable-domain x)))))
```

Conventions—IV

Definition: A *type predicate* is a one argument procedure that returns a Boolean value indicating whether its argument is a member of a specified type.

Convention: A type predicate for *type* is named `type?`.

Examples: `number?`, `boolean?`

Conventions—V

Definition: A *type coercion* is a one-argument procedure that maps arguments of $type_1$ to results of $type_2$.

Convention: A type coercion that maps $type_1$ to $type_2$ is named `type`$_1$`->type`$_2$.

Examples: `exact->inexact`, `integer->char`, `list->string`

Case Insensitivity

```
(define (factorial n)
 (if (zero? n) 1 (* n (factorial (- n 1)))))

(DEFINE (FACTORIAL N)
 (IF (ZERO? N) 1 (* N (FACTORIAL (- N 1)))))

(Define (Factorial N)
 (If (Zero? N) 1 (* N (Factorial (- N 1)))))
```

`zero?`, `ZERO?`, `Zero?`, and even `zErO?` are all equivalent and refer to the same procedure.

Conventions—VI

Convention: Variable (and procedure) names are written in lower case, not upper case or capitalized.

Examples:	`number?`
not:	`NUMBER?`, `Number?`

Conventions—VII

Convention: Use '-' to separate multiple words in variable (and procedure) names, not '_', capitalization, or no separation.

Examples: `invert-matrix`
not: `invert_matrix`, `invertMatrix`, `invertmatrix`

Conventions—VIII

Convention: Use full English words in variable (and procedure) names.

Examples: `invert-matrix`
not: `inv-mat`, `invmat`

Conventions—IX

Convention: Limit line length to 79 characters so there is no wraparound.

Conventions—X

Convention: Use a tool to properly indent your code automatically.

Example:

```
(define (factorial n)
 (if (zero? n)
     1
     (* n (factorial (- n 1)))))
```

not:

```
(define (factorial n)
   (if (zero? n)
                 1
         (* n (factorial (- n 1)))))
```

Conventions—XI

Convention: Do not put whitespace after '`(`' or before '`)`'.

Example:

```
(define (factorial n)
 (if (zero? n)
     1
     (* n (factorial (- n 1)))))
```

not:

```
(
 define (factorial n) (
 if (zero? n)
    1
    (* n ( factorial (- n 1) ))
 )
)
```

Conventions—XII

Convention: Put a single space between elements of a list on the same line.

Examples: `(a b c)`, `(() ())`

not: `(a   b   c)`, `(()())`, `(()    ())`

Conventions—XIII

Convention: The alternate of an `if` expression is optional. But it is bad practice to leave it out. Use `when` or `unless` instead.

bad:

```
(if (bound? domain-variable)
    (place-queen i (binding domain-variable)))
```

instead:

```
(when (bound? domain-variable)
 (place-queen i (binding domain-variable)))
```

Conventions—XIV

Convention: The `else` clause of a `cond` expression is optional. But it is bad practice to leave it out. Put in a dummy `else` clause.

bad:

```
(define (signum n)
 (cond ((negative? n) -1)
       ((zero? n) 0)
       ((positive? n) 1)))
```

instead:

```
(define (signum n)
 (cond ((negative? n) -1)
       ((zero? n) 0)
       ((positive? n) 1)
       (else (panic "This shouldn't happen"))))
```