Comparison Functions

- Boolean functions (by convention) end with a `?`.
- We can discriminate between different kinds of numbers:

  ```scheme
  > (complex? 3+4i)
  #t
  > (complex? 3)
  #t
  > (real? 3)
  #t
  > (real? -2.5+0.0i)
  #t
  > (rational? 6/10)
  #t
  > (rational? 6/3)
  #t
  > (integer? 3+0i)
  #t
  > (integer? 3.0)
  #t
  > (integer? 8/4)
  #t
  ```

Tests on Numbers

- Several of the comparison functions can take multiple arguments.
- `(<= 4 5 6 7 9 234)` returns true since the numbers are monotonically increasing.

  ```scheme
  > (< 4 5)
  true
  > (< 4 5 6 7 9 234)
  true
  > (> 5 2 1 3)
  false
  > (= 1 1 1 1 1)
  true
  > (<= 1 2 2 2 3)
  true
  ```
Tests on Numbers…

> (>= 5 5)
true
> (zero? 5)
false
> (positive? 5)
true
> (negative? 5)
false
> (odd? 5)
true
> (even? 5)
false

Conditionals — If

- If the test-expression evaluates to \#f (False) return the value of the then-expression, otherwise return the value of the else-expression:

  (if test-expression
   then-expression
   else-expression)

- Up to language level “Advanced Student” if-expressions must have two parts.

- Set the language level to **Standard (R5RS)** to get the standard Scheme behavior, where the else-expression is optional.

Conditionals — If...

> (define x 5)
> (if (= x 5) 2 4)
2
> (if (< x 3)
   (display "hello")
   (display "bye"))
bye
> (display
   (if (< x 3) "hello" "bye"))
bye

If it’s not False (\#f), it’s True (\#t)

- Any value that is not false, is interpreted as true.

- NOTE: In DrScheme this depends on which language level you set. Up to “Advanced Student”, the test-expression of an if must be either \#t or \#f.

- Set the language level to **Standard (R5RS)** to get the standard Scheme behavior:

  > (if 5 "hello" "bye")
  "hello"
  > (if #f "hello" "bye")
  "bye"
  > (if #f "hello")
  > (if #t "hello")
  "hello"
Boolean Operators

- and and or can take multiple arguments.
- and returns true if none of its arguments evaluate to False.
- or returns true if any of its arguments evaluates to True.

> (and (< 3 5) (odd? 5) (inexact? (cos 32)))
#t
> (or (even? 5) (zero? (- 5 5)))
#t
> (not 5)
#f
> (not #t)
#f

Defining Boolean Functions

- We can define our own boolean functions:

```
(define (big-number? n)
  (> n 10000000))
```

> (big-number? 5)
#f
> (big-number? 384783274832748327)
#t

Conditionals — cond

- cond is a generalization of if:

```
(cond
  (cond-expression1 result-expression1)
  (cond-expression2 result-expression2)
  ...
  (else else-expression))
```

- Each cond-expression_i is evaluated in turn, until one evaluates to not False.

> (cond
  ((< 2 3) 4)
  ((= 2 3) 5)
  (else 6))
4

In general, any value that is not #f is considered true.
and and or evaluate their arguments from left to right, and stop as soon as they know the final result.
The last value evaluated is the one returned.

> (and "hello")
"hello"
> (and "hello" "world")
"world"
> (or "hello" "world")
"hello"
Conditionals — cond...

To make this a bit more readable, we use square brackets around the cond-clauses:

```
(cond
    [cond-expr1 result-expr1]
    [cond-expr2 result-expr2]
    ...
    [else else-expression])
```

```
> (cond [#f 5] [#t 6])
6
> (cond
    [(= 4 5) "hello"]
    [(> 4 5) "goodbye"]
    [(< 4 5) "see ya!"])
"see ya!"
```

Conditionals — case...

case is like Java/C’s switch statement:

```
(case key
    [(expr1 expr2 ...) result-expr1]
    [(expr11 expr11 ...) result-expr2]
    ...
    (else else-expression))
```

The key is evaluated once, and compared against each cond-expr in turn, and the corresponding result-expr is returned.

```
> (case 5 [(2 3) "hello"] [(4 5) "bye"]) "bye"
```

Sequencing

To do more than one thing in sequence, use begin:

```
(begin arg1 arg2 ...)
```

```
> (begin
    (display "the meaning of life=")
    (display (* 6 7))
    (newline)
)
the meaning of life=42
```
Examples — \( n! \)

Write the factorial function \( n! \):

\[
\text{(define (! n)} \\
\text{\quad (cond)} \\
\text{\quad \quad [(zero? n) 1] } \\
\text{\quad \quad [else (* n (! (- n 1)))]]} \\
\text{\quad )} \\
\text{)} \\
\text{> (! 5)} \\
120
\]

Examples — \( \binom{n}{r} \)

Write the \( \binom{n}{r} \) function in Scheme:

\[
\binom{n}{r} = \frac{n!}{r! \cdot (n-r)!}
\]

Use the factorial function from the last slide.

\[
\text{(define (choose n r)} \\
\text{\quad (/ (! n) (\ast (! r) (! (- n r)))))} \\
\text{\quad )} \\
\text{)} \\
\text{> (choose 5 2)} \\
10
\]

Examples — (sum m n)

Write a function \( \text{(sum m n)} \) that returns the sum of the integers between \( m \) and \( n \), inclusive.

\[
\text{(define (sum m n)} \\
\text{\quad (cond)} \\
\text{\quad \quad [(= m n) m] } \\
\text{\quad \quad [else (+ m (sum (+ 1 m) n))]}} \\
\text{\quad )} \\
\text{)} \\
\text{> (sum 1 2)} \\
3 \\
\text{> (sum 1 4)} \\
10
\]

Examples — Ackermann’s function

Implement Ackermann’s function:

\[
\begin{align*}
A(1, j) &= 2j &\text{for } j \geq 1 \\
A(i, 1) &= A(i-1, 2) &\text{for } i \geq 2 \\
A(i, j) &= A(i-1, A(i, j-1)) &\text{for } i, j \geq 2
\end{align*}
\]

\[
\text{(define (A i j)} \\
\text{\quad (cond)} \\
\text{\quad \quad [(and (= i 1) (>= j 1)) (\ast 2 j)]} \\
\text{\quad \quad [(and (> i 2) (= j 1)) (A(- i 1) 2)]} \\
\text{\quad \quad [(and (> i 2) (> j 2)) (A(- i 1) (A i (- j 1)))]}} \\
\text{\quad )} \\
\text{)}
\]
Examples — Ackermann’s function...

Ackermann’s function grows very quickly:

> (A 1 1)
2
> (A 3 2)
512
> (A 3 3)
1561585988519419914804999641169225
4958731641184786755447122887443528
0601470939536037485963338068553800
6371637297210170750776562389313989
2867298012168192

Scheme so Far

Unlike languages like Java and C which are statically typed (we describe in the program text what type each variable is) Scheme is dynamically typed. We can test at runtime what particular type of number an atom is:

- (complex? arg), (real? arg)
- (rational? arg), (integer? arg)

Tests on numbers:

- (< arg1, arg2), (> arg1, arg2)
- (= arg1, arg2), (<= arg1, arg2)
- (>= arg1, arg2), (zero? arg)
- (positive? arg), (negative? arg)
- (odd? arg), (even? arg)

Scheme so Far...

Unlike many other languages like Java which are statement-oriented, Scheme is expression-oriented. That is, every construct (even if, cond, etc) return a value. The if-expression returns the value of the then-expr or the else-expr:

(if test-expr then-expr else-expr)

depending on the value of the test-expr.

Scheme so Far...

The cond-expression evaluates its guards until one evaluates to non-false. The corresponding value is returned:

(cond
  (guard1 value1)
  (guard2 value2)
  ...
  (else else-expr))
The `case`-expression evaluates `key`, finds the first matching expression, and returns the corresponding result:

```
(case key
  [(expr₁ expr₂ ...) result-expr₁]
  [(expr₁₁ expr₁₂ ...) result-expr₂]
  ...
  (else else-expr))
```

and `and` or `or` take multiple arguments, evaluate their results left-to-right until the outcome can be determined (for `or` when the first non-`false`, for `and` when the first `false` is found), and returns the last value evaluated.