Getting Started with the CLISP Common Lisp Interpreter

Getting Started with the CLISP Common Lisp Interpreter

-
-
-
-
-

Getting Started with the CLISP Common Lisp Interpreter

- While logged in to **euclid** or **venus**, enter

  ```
  cl
  ```

  at the shell prompt to start the CLISP interpreter.
-
-
-
-

Getting Started with the CLISP Common Lisp Interpreter

- While logged in to **euclid** or **venus**, enter

  ```
  cl
  ```

  at the shell prompt to start the CLISP interpreter.
- CLISP's prompt is **[*n*]>** (e.g., **[1]>**, **[2]>**, **[3]>**, ...), where *n* is a count of the number of prompts that have been displayed so far.
-
-
-

Getting Started with the CLISP Common Lisp Interpreter

- While logged in to **euclid** or **venus**, enter

  cl

  at the shell prompt to start the CLISP interpreter.
- CLISP's prompt is **[*n*]>** (e.g., **[1]>**, **[2]>**, **[3]>**, ...), where *n* is a count of the number of prompts that have been displayed so far.
- At any prompt, you can exit from CLISP by typing CTRL d or by entering (exit) [*including* the parentheses!].
-
-

Getting Started with the CLISP Common Lisp Interpreter

- While logged in to **euclid** or **venus**, enter

  ```
  cl
  ```

  at the shell prompt to start the CLISP interpreter.
- CLISP's prompt is **[*n*]>** (e.g., **[1]>**, **[2]>**, **[3]>**, ...), where *n* is a count of the number of prompts that have been displayed so far.
- At any prompt, you can exit from CLISP by typing CTRL d or by entering (exit) [*including* the parentheses!].
- At any prompt, you can enter a Lisp expression to be evaluated. Lisp will ***read*** in your expression, ***evaluate*** it, and then ***print*** the expression's value.
-

Getting Started with the CLISP Common Lisp Interpreter

- While logged in to **euclid** or **venus**, enter

  ```
  cl
  ```

  at the shell prompt to start the CLISP interpreter.

- CLISP's prompt is **[*n*]>** (e.g., **[1]>**, **[2]>**, **[3]>**, ...), where *n* is a count of the number of prompts that have been displayed so far.

- At any prompt, you can exit from CLISP by typing CTRL d or by entering (exit) [*including* the parentheses!].

- At any prompt, you can enter a Lisp expression to be evaluated. Lisp will ***read*** in your expression, ***evaluate*** it, and then ***print*** the expression's value.

- Lisp expressions are written in a special notation:

  Java: Lisp:

  Java: Lisp:

  Java: Lisp:

Getting Started with the CLISP Common Lisp Interpreter

- While logged in to **euclid** or **venus**, enter

  ```
  cl
  ```

  at the shell prompt to start the CLISP interpreter.

- CLISP's prompt is **[*n*]>** (e.g., **[1]>**, **[2]>**, **[3]>**, ...), where *n* is a count of the number of prompts that have been displayed so far.

- At any prompt, you can exit from CLISP by typing CTRL d or by entering (exit) [*including* the parentheses!].

- At any prompt, you can enter a Lisp expression to be evaluated. Lisp will ***read*** in your expression, ***evaluate*** it, and then ***print*** the expression's value.

- Lisp expressions are written in a special notation:

Java: 3 - 4.1	Lisp: (- 3 4.1)
Java:	Lisp:
Java:	Lisp:

Getting Started with the CLISP Common Lisp Interpreter

- While logged in to **euclid** or **venus**, enter

  cl

  at the shell prompt to start the CLISP interpreter.

- CLISP's prompt is **[*n*]>** (e.g., **[1]>**, **[2]>**, **[3]>**, ...), where *n* is a count of the number of prompts that have been displayed so far.

- At any prompt, you can exit from CLISP by typing CTRL d or by entering (exit) [*including* the parentheses!].

- At any prompt, you can enter a Lisp expression to be evaluated. Lisp will ***read*** in your expression, ***evaluate*** it, and then ***print*** the expression's value.

- Lisp expressions are written in a special notation:

Java: 3 - 4.1	Lisp: (- 3 4.1)
Java: 3 - 4.1 + 2	Lisp: (+ (- 3 4.1) 2)
Java:	Lisp:

Getting Started with the CLISP Common Lisp Interpreter

- While logged in to **euclid** or **venus**, enter

  ```
  cl
  ```

  at the shell prompt to start the CLISP interpreter.

- CLISP's prompt is **[*n*]>** (e.g., **[1]>**, **[2]>**, **[3]>**, ...), where *n* is a count of the number of prompts that have been displayed so far.

- At any prompt, you can exit from CLISP by typing CTRL d or by entering (exit) [*including* the parentheses!].

- At any prompt, you can enter a Lisp expression to be evaluated. Lisp will ***read*** in your expression, ***evaluate*** it, and then ***print*** the expression's value.

- Lisp expressions are written in a special notation:

Java: 3 - 4.1	Lisp: (- 3 4.1)
Java: 3 - 4.1 + 2	Lisp: (+ (- 3 4.1) 2)
Java: Math.sqrt(3*4.1)	Lisp: (sqrt (* 3 4.1))

Getting Started with the CLISP Common Lisp Interpreter

- At any prompt, you can enter a Lisp expression to be evaluated. Lisp will ***read*** in your expression, ***evaluate*** it, and then ***print*** the expression's value.
- Lisp expressions are written in a special notation:

```
Java: 3 – 4.1                      Lisp: (– 3 4.1)
Java: 3 – 4.1 + 2                  Lisp: (+ (– 3 4.1) 2)
Java: Math.sqrt(3*4.1)             Lisp: (sqrt (* 3 4.1))
```

Getting Started with the CLISP Common Lisp Interpreter

- At any prompt, you can enter a Lisp expression to be evaluated. Lisp will ***read*** in your expression, ***evaluate*** it, and then ***print*** the expression's value.
- Lisp expressions are written in a special notation:

```
Java: 3 – 4.1                     Lisp: (– 3 4.1)
Java: 3 – 4.1 + 2                 Lisp: (+ (– 3 4.1) 2)
Java: Math.sqrt(3*4.1)            Lisp: (sqrt (* 3 4.1))
```

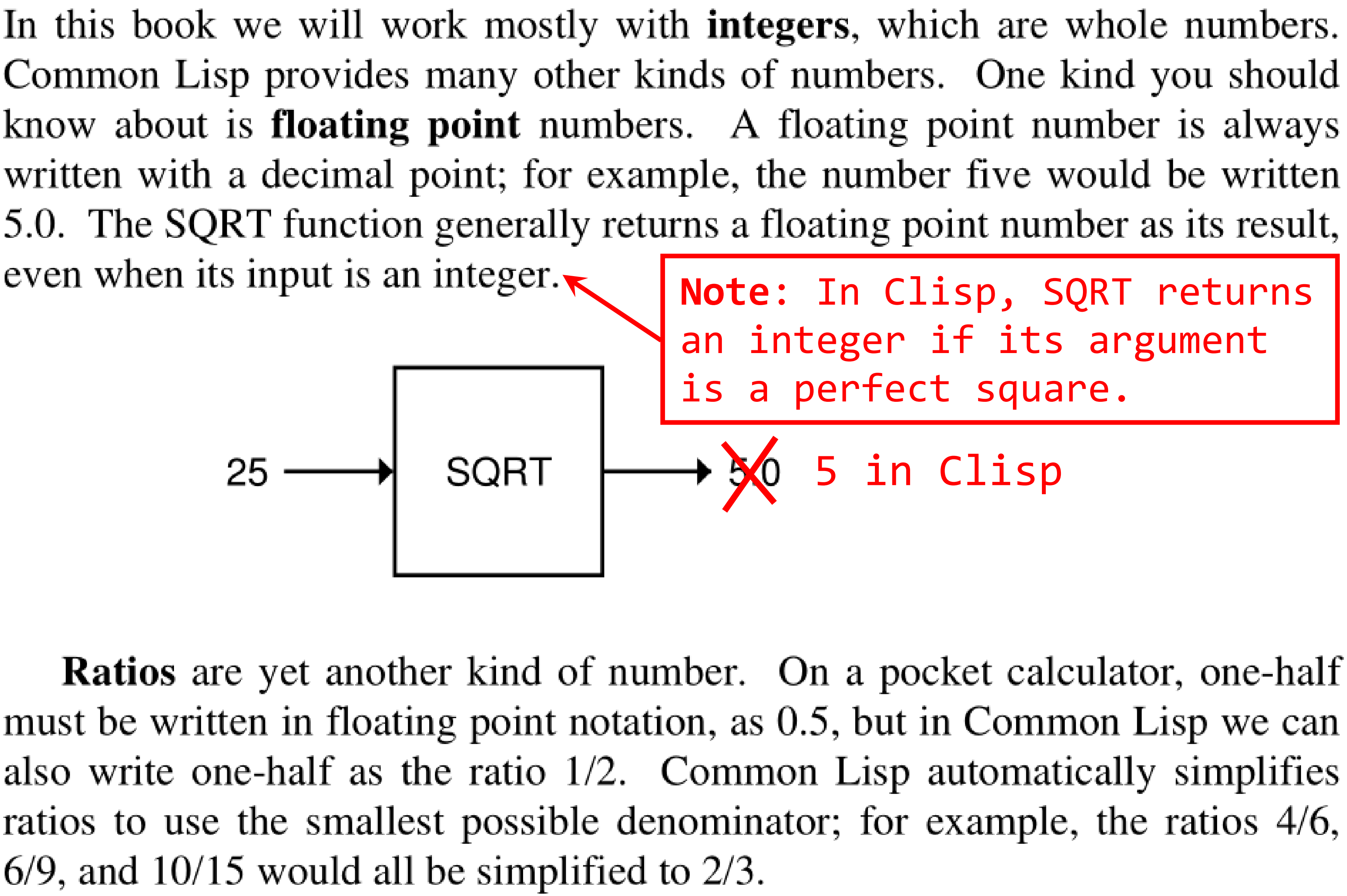
Getting Started with the CLISP Common Lisp Interpreter

- At any prompt, you can enter a Lisp expression to be evaluated. Lisp will ***read*** in your expression, ***evaluate*** it, and then ***print*** the expression's value.
- Lisp expressions are written in a special notation:

Java: 3 – 4.1	Lisp: (– 3 4.1)
Java: 3 – 4.1 + 2	Lisp: (+ (– 3 4.1) 2)
Java: Math.sqrt(3*4.1)	Lisp: (sqrt (* 3 4.1))

  - If an integer that is a perfect square (e.g., 9) is passed as argument to CLISP's sqrt, its ***integer*** square root is returned; in some other implementations of Common Lisp, a floating point result is returned.

From p. 3 of Touretzky:



In this book we will work mostly with **integers**, which are whole numbers. Common Lisp provides many other kinds of numbers. One kind you should know about is **floating point** numbers. A floating point number is always written with a decimal point; for example, the number five would be written 5.0. The SQRT function generally returns a floating point number as its result, even when its input is an integer.

Note: In Clisp, SQRT returns an integer if its argument is a perfect square.

5 in Clisp

Ratios are yet another kind of number. On a pocket calculator, one-half must be written in floating point notation, as 0.5, but in Common Lisp we can also write one-half as the ratio 1/2. Common Lisp automatically simplifies ratios to use the smallest possible denominator; for example, the ratios 4/6, 6/9, and 10/15 would all be simplified to 2/3.

Getting Started with the CLISP Common Lisp Interpreter

- At any prompt, you can enter a Lisp expression to be evaluated. Lisp will ***read*** in your expression, ***evaluate*** it, and then ***print*** the expression's value.
- Lisp expressions are written in a special notation:

Java: 3 – 4.1	Lisp: (– 3 4.1)
Java: 3 – 4.1 + 2	Lisp: (+ (– 3 4.1) 2)
Java: Math.sqrt(3*4.1)	Lisp: (sqrt (* 3 4.1))

  - If an integer that is a perfect square (e.g., 9) is passed as argument to CLISP's sqrt, its ***integer*** square root is returned; in some other implementations of Common Lisp, a floating point result is returned.

Getting Started with the CLISP Common Lisp Interpreter

- At any prompt, you can enter a Lisp expression to be evaluated. Lisp will ***read*** in your expression, ***evaluate*** it, and then ***print*** the expression's value.
- Lisp expressions are written in a special notation:

Java: 3 – 4.1	Lisp: (– 3 4.1)
Java: 3 – 4.1 + 2	Lisp: (+ (– 3 4.1) 2)
Java: Math.sqrt(3*4.1)	Lisp: (sqrt (* 3 4.1))

  - If an integer that is a perfect square (e.g., 9) is passed as argument to CLISP's sqrt, its ***integer*** square root is returned; in some other implementations of Common Lisp, a floating point result is returned.
- If evaluation of an expression produces an error, then CLISP prints an error message followed by a **Break ... >** prompt:

Getting Started with the CLISP Common Lisp Interpreter

- At any prompt, you can enter a Lisp expression to be evaluated. Lisp will ***read*** in your expression, ***evaluate*** it, and then ***print*** the expression's value.
- Lisp expressions are written in a special notation:

Java: 3 – 4.1	Lisp: (– 3 4.1)
Java: 3 – 4.1 + 2	Lisp: (+ (– 3 4.1) 2)
Java: Math.sqrt(3*4.1)	Lisp: (sqrt (* 3 4.1))

  - If an integer that is a perfect square (e.g., 9) is passed as argument to CLISP's sqrt, its ***integer*** square root is returned; in some other implementations of Common Lisp, a floating point result is returned.
- If evaluation of an expression produces an error, then CLISP prints an error message followed by a **Break ... >** prompt: You can enter **:q** at a **Break ... >** prompt to get back to the regular [*n*]> prompt!

Getting Started with the CLISP Common Lisp Interpreter

- At any prompt, you can enter a Lisp expression to be evaluated. Lisp will ***read*** in your expression, ***evaluate*** it, and then ***print*** the expression's value.
- Lisp expressions are written in a special notation:

Java: 3 – 4.1	Lisp: (– 3 4.1)
Java: 3 – 4.1 + 2	Lisp: (+ (– 3 4.1) 2)
Java: Math.sqrt(3*4.1)	Lisp: (sqrt (* 3 4.1))

  - If an integer that is a perfect square (e.g., 9) is passed as argument to CLISP's sqrt, its ***integer*** square root is returned; in some other implementations of Common Lisp, a floating point result is returned.

- If evaluation of an expression produces an error, then CLISP prints an error message followed by a **Break** ... **>** prompt: You can enter **:q** at a **Break** ... **>** prompt to get back to the regular [*n*]> prompt!

  Example: The function sqrt expects just one argument, so evaluation of (sqrt 4 5) produces a **Break** ... **>**.

Assigning Values to Global Variables

Assigning Values to Global Variables

- **SETF** can be used to assign a value to a variable:

Assigning Values to Global Variables

- **SETF** can be used to assign a value to a variable:

  evaluation of (setf x *expr*) has the side-effect of assigning *expr*'s value to the variable x.

Assigning Values to Global Variables

- **SETF** can be used to assign a value to a variable: The value of the Lisp expression (setf x *expr*) is the value of *expr*--e.g., the value of (+ (setf x 3) 5) is 8--but evaluation of (setf x *expr*) has the side-effect of assigning *expr*'s value to the variable x.

Assigning Values to Global Variables

- **SETF** can be used to assign a value to a variable: The value of the Lisp expression (setf x *expr*) is the value of *expr*--e.g., the value of (+ (setf x 3) 5) is 8--but evaluation of (setf x *expr*) has the side-effect of assigning *expr*'s value to the variable x.
- Thus (setf x ...) is analogous to a C++ or Java expression of the form (x = ...)!

Assigning Values to Global Variables

- **SETF** can be used to assign a value to a variable: The value of the Lisp expression (setf x *expr*) is the value of *expr*--e.g., the value of (+ (setf x 3) 5) is 8--but evaluation of (setf x *expr*) has the side-effect of assigning *expr*'s value to the variable x.
- Thus (setf x ...) is analogous to a C++ or Java expression of the form (x = ...)!
- Once a variable has been assigned a value, the variable can be used to represent that value in subsequent expressions.

Assigning Values to Global Variables

- **SETF** can be used to assign a value to a variable: The value of the Lisp expression (setf x *expr*) is the value of *expr*--e.g., the value of (+ (setf x 3) 5) is 8--but evaluation of (setf x *expr*) has the side-effect of assigning *expr*'s value to the variable x.
- Thus (setf x ...) is analogous to a C++ or Java expression of the form (x = ...)!
- Once a variable has been assigned a value, the variable can be used to represent that value in subsequent expressions.
- **IMPORTANT**: SETF is ***not*** used in pure functional programming,

Assigning Values to Global Variables

- **SETF** can be used to assign a value to a variable: The value of the Lisp expression (setf x *expr*) is the value of *expr*--e.g., the value of (+ (setf x 3) 5) is 8--but evaluation of (setf x *expr*) has the side-effect of assigning *expr*'s value to the variable x.

- Thus (setf x ...) is analogous to a C++ or Java expression of the form (x = ...)!

- Once a variable has been assigned a value, the variable can be used to represent that value in subsequent expressions.

- **IMPORTANT**: SETF is ***not*** used in pure functional programming, so the Lisp functions you write when doing programming assignments or answering exam questions must ***not*** use SETF!

Assigning Values to Global Variables

- **SETF** can be used to assign a value to a variable: The value of the Lisp expression (setf x *expr*) is the value of *expr*--e.g., the value of (+ (setf x 3) 5) is 8--but evaluation of (setf x *expr*) has the side-effect of assigning *expr*'s value to the variable x.

- **IMPORTANT**: SETF is ***not*** used in pure functional programming, so the Lisp functions you write when doing programming assignments or answering exam questions must ***not*** use SETF!

Assigning Values to Global Variables

- **SETF** can be used to assign a value to a variable: The value of the Lisp expression (setf x *expr*) is the value of *expr*--e.g., the value of (+ (setf x 3) 5) is 8--but evaluation of (setf x *expr*) has the side-effect of assigning *expr*'s value to the variable x.

- **IMPORTANT**: SETF is ***not*** used in pure functional programming, so the Lisp functions you write when doing programming assignments or answering exam questions must ***not*** use SETF!

Assigning Values to Global Variables

- **SETF** can be used to assign a value to a variable: The value of the Lisp expression (setf x *expr*) is the value of *expr*--e.g., the value of (+ (setf x 3) 5) is 8--but evaluation of (setf x *expr*) has the side-effect of assigning *expr*'s value to the variable x.

- **IMPORTANT**: SETF is ***not*** used in pure functional programming, so the Lisp functions you write when doing programming assignments or answering exam questions must ***not*** use SETF!

- You may use SETF when ***testing*** your functions:

Assigning Values to Global Variables

- **SETF** can be used to assign a value to a variable: The value of the Lisp expression (setf x *expr*) is the value of *expr*--e.g., the value of (+ (setf x 3) 5) is 8--but evaluation of (setf x *expr*) has the side-effect of assigning *expr*'s value to the variable x.

- **IMPORTANT**: SETF is ***not*** used in pure functional programming, so the Lisp functions you write when doing programming assignments or answering exam questions must ***not*** use SETF!

- You may use SETF when ***testing*** your functions:

  For example, if you plan to use $2^{31} - 1$ as a test argument value several times, then you can use SETF to store $2^{31} - 1$ in a variable that will be used as the actual argument each time.

Ratios

Ratios (a type of number that represents fractions)

- Unlike C++ and Java, Common Lisp has a built-in data type called ***ratio*** that represents (positive and negative, proper and improper) fractions ***exactly***, with no rounding error. Examples:

Ratios (a type of number that represents fractions)

- Unlike C++ and Java, Common Lisp has a built-in data type called ***ratio*** that represents (positive and negative, proper and improper) fractions ***exactly***, with no rounding error. Examples:

  6/8 represents the positive proper fraction $\frac{6}{8} = \frac{3}{4}$.

Ratios (a type of number that represents fractions)

- Unlike C++ and Java, Common Lisp has a built-in data type called ***ratio*** that represents (positive and negative, proper and improper) fractions ***exactly***, with no rounding error. Examples:

  6/8 represents the positive proper fraction $\frac{6}{8} = \frac{3}{4}$.

  -5/4 represents the negative improper fraction $\frac{-5}{4}$.

Ratios (a type of number that represents fractions)

- Unlike C++ and Java, Common Lisp has a built-in data type called ***ratio*** that represents (positive and negative, proper and improper) fractions ***exactly***, with no rounding error. Examples:

  6/8 represents the positive proper fraction $\frac{6}{8} = \frac{3}{4}$.

  -5/4 represents the negative improper fraction $\frac{-5}{4}$.

  - Ratios are always ***printed in lowest terms*** (but they need not be written in lowest terms): You can write 6/8, but this ratio would be printed as 3/4.

Ratios (a type of number that represents fractions)

- Unlike C++ and Java, Common Lisp has a built-in data type called ***ratio*** that represents (positive and negative, proper and improper) fractions ***exactly***, with no rounding error. Examples:

  6/8 represents the positive proper fraction $\frac{6}{8} = \frac{3}{4}$.

  -5/4 represents the negative improper fraction $\frac{-5}{4}$.

  - Ratios are always ***printed in lowest terms*** (but they need not be written in lowest terms): You can write 6/8, but this ratio would be printed as 3/4.
  - If m and n are integers, n is not 0, and n does not divide m, then the value of (/ m n) is a ratio. For example, the value of (/ 12 9) is 12/9 = 4/3.

Ratios (a type of number that represents fractions)

- Unlike C++ and Java, Common Lisp has a built-in data type called ***ratio*** that represents (positive and negative, proper and improper) fractions ***exactly***, with no rounding error. Examples:

  6/8 represents the positive proper fraction $\frac{6}{8} = \frac{3}{4}$.

  -5/4 represents the negative improper fraction $\frac{-5}{4}$.

  - Ratios are always ***printed in lowest terms*** (but they need not be written in lowest terms): You can write 6/8, but this ratio would be printed as 3/4.
  - If m and n are integers, n is not 0, and n does not divide m, then the value of (/ m n) is a ratio. For example, the value of (/ 12 9) is 12/9 = 4/3.
  - There is no space before or after the / in a ratio: 5/7 ***cannot*** be written as 5 /7 or 5/ 7.

Ratios (a type of number that represents fractions)

- Unlike C++ and Java, Common Lisp has a built-in data type called ***ratio*** that represents fractions ***exactly***, with no rounding error.

  - Ratios are always ***printed in lowest terms*** (but they need not be written in lowest terms): You can write 6/8, but this ratio would be printed as 3/4.
  - If m and n are integers, n is not 0, and n does not divide m, then the value of (/ m n) is a ratio. For example, the value of (/ 12 9) is 12/9 = 4/3.
  - There is no space before or after the / in a ratio: 5/7 ***cannot*** be written as 5 /7 or 5/ 7.

Ratios (a type of number that represents fractions)

- Unlike C++ and Java, Common Lisp has a built-in data type called ***ratio*** that represents fractions ***exactly***, with no rounding error.
  - Ratios are always ***printed in lowest terms*** (but they need not be written in lowest terms): You can write 6/8, but this ratio would be printed as 3/4.
  - If m and n are integers, n is not 0, and n does not divide m, then the value of (/ m n) is a ratio. For example, the value of (/ 12 9) is 12/9 = 4/3.
  - There is no space before or after the / in a ratio: 5/7 ***cannot*** be written as 5 /7 or 5/ 7.

Ratios (a type of number that represents fractions)

- Unlike C++ and Java, Common Lisp has a built-in data type called ***ratio*** that represents fractions ***exactly***, with no rounding error.
  - Ratios are always ***printed in lowest terms*** (but they need not be written in lowest terms): You can write 6/8, but this ratio would be printed as 3/4.
  - If m and n are integers, n is not 0, and n does not divide m, then the value of (/ m n) is a ratio. For example, the value of (/ 12 9) is 12/9 = 4/3.
  - There is no space before or after the / in a ratio: 5/7 ***cannot*** be written as 5 /7 or 5/ 7.

Ratios (a type of number that represents fractions)

- Unlike C++ and Java, Common Lisp has a built-in data type called ***ratio*** that represents fractions ***exactly***, with no rounding error.
  - Ratios are always ***printed in lowest terms*** (but they need not be written in lowest terms): You can write 6/8, but this ratio would be printed as 3/4.
  - If m and n are integers, n is not 0, and n does not divide m, then the value of (/ m n) is a ratio. For example, the value of (/ 12 9) is 12/9 = 4/3.
  - There is no space before or after the / in a ratio: 5/7 ***cannot*** be written as 5 /7 or 5/ 7.
  - In Common Lisp, a number is said to be ***rational*** if it is *either* an integer *or* a ratio.

Ratios (a type of number that represents fractions)

- Unlike C++ and Java, Common Lisp has a built-in data type called ***ratio*** that represents fractions ***exactly***, with no rounding error.
  - Ratios are always ***printed in lowest terms*** (but they need not be written in lowest terms): You can write 6/8, but this ratio would be printed as 3/4.
  - If m and n are integers, n is not 0, and n does not divide m, then the value of (/ m n) is a ratio. For example, the value of (/ 12 9) is 12/9 = 4/3.
  - There is no space before or after the / in a ratio: 5/7 ***cannot*** be written as 5 /7 or 5/ 7.
  - In Common Lisp, a number is said to be ***rational*** if it is *either* an integer *or* a ratio.
  - The functions +, -, *, and / accept rational and floating point argument values:

Ratios (a type of number that represents fractions)

- Unlike C++ and Java, Common Lisp has a built-in data type called ***ratio*** that represents fractions ***exactly***, with no rounding error.
  - Ratios are always ***printed in lowest terms*** (but they need not be written in lowest terms): You can write 6/8, but this ratio would be printed as 3/4.
  - If m and n are integers, n is not 0, and n does not divide m, then the value of (/ m n) is a ratio. For example, the value of (/ 12 9) is 12/9 = 4/3.
  - There is no space before or after the / in a ratio: 5/7 ***cannot*** be written as 5 /7 or 5/ 7.
  - In Common Lisp, a number is said to be ***rational*** if it is *either* an integer *or* a ratio.
  - The functions +, -, *, and / accept rational and floating point argument values: If each argument value is rational, the returned result will also be rational; otherwise, the result will be a floating point number.