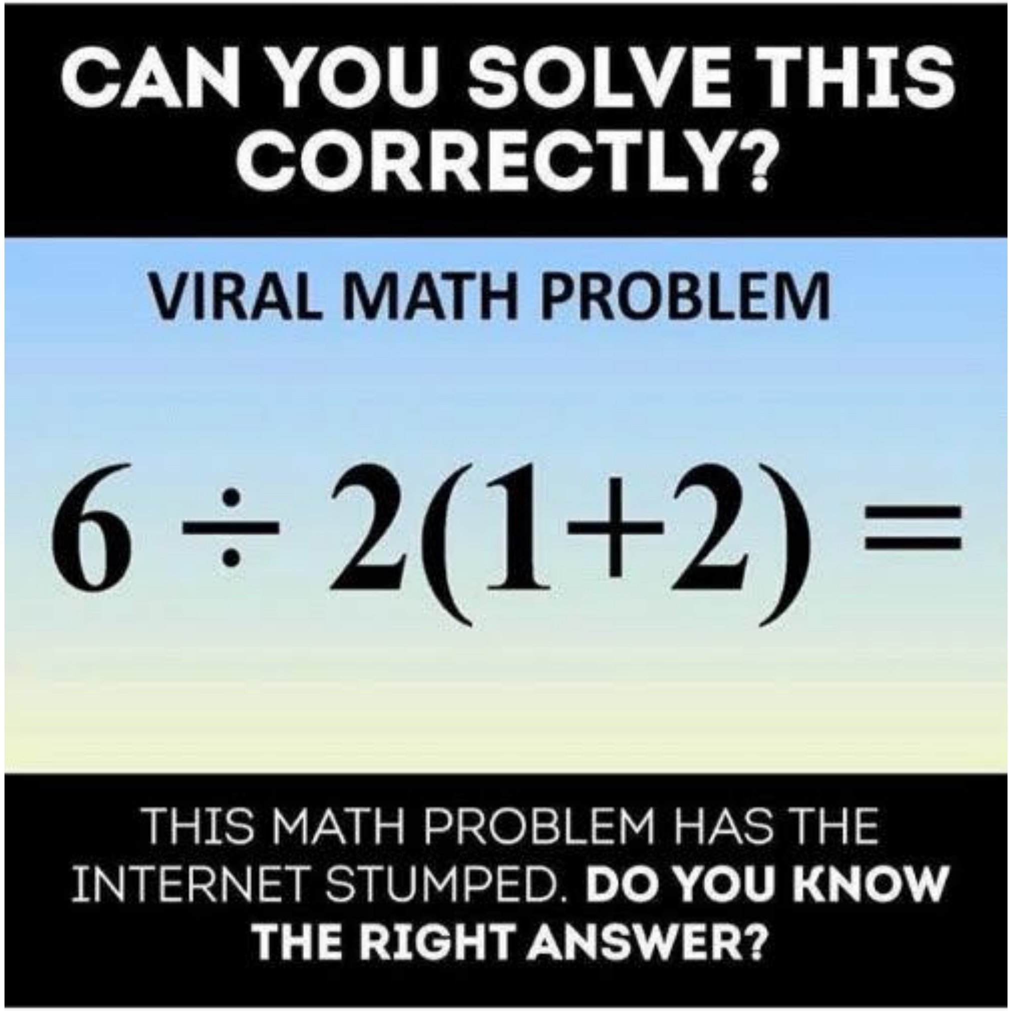
Values and expressions

CS135 Lecture 01

L01.0 Prefix notation



CAN YOU SOLVE THIS CORRECTLY?

VIRAL MATH PROBLEM

$$6 \div 2(1+2) =$$

THIS MATH PROBLEM HAS THE INTERNET STUMPED. **DO YOU KNOW THE RIGHT ANSWER?**

?

“Viral math problem”

This “problem” seems confusing because it mixes notational systems, only uses parentheses for some subexpressions, and requires you to remember the rules for ordering operations.

$$
\begin{aligned}
6 \div 2(1 + 2) \quad &\Rightarrow ((6 / 2) \cdot (1 + 2)) \\
&\Rightarrow (3 \cdot (1 + 2)) \\
&\Rightarrow (3 \cdot 3) \\
&\Rightarrow 9
\end{aligned}
$$

ORDER OF OPERATIONS
PARENTHESES, EXPONENTS, DIVISION & MULTIPLICATION, ADDITION & SUBTRACTION
TRADITIONAL: PLEASE EXCUSE MY DEAR AUNT SALLY
RIGHT THERE.
PLEASE EMAIL MY DAD A SHARK

https://www.xkcd.com/992/

Prefix notation

In elementary and high school, you used **infix notation** for mathematical expressions:

((6 / 2) · (1 + 2))

Racket uses **prefix notation** for expressions.

- All expressions are surrounded by parenthesis.
- The operator goes first.
- Racket uses "`*`" instead of "·" for multiplication.

```
(* (/ 6 2) (+ 1 2))
```

Prefix notation provides a uniform notation for Racket functions, operators, etc.

Evaluating a racket expression

We use a process of substitution, just as with our mathematical expressions. Each step is indicated using the ‘yields’ symbol ⇒

```
(* (/ 6 2)(+ 1 2))
⇒ (* 3 (+ 1 2))
⇒ (* 3 3)
⇒ 9
```

The substitution process repeatedly simplifies the program. At each step, the result is a valid (but simpler) Racket program. It eventually simplifies to a value.

A **substitution step** finds the **leftmost/inner subexpression,** with no parentheses inside, and rewrites it by replacing the subexpression by its value.

Substitution steps

```
(- (+ 9 (* (/ 6 (+ 1 1)) 3)) 2)  ⇒ (- (+ 9 (* (/ 6 2) 3)) 2)
                                 ⇒ (- (+ 9 (* 3 3)) 2)
                                 ⇒ (- (+ 9 9) 2)
                                 ⇒ (- 18 2)
                                 ⇒ 16
```

Basic arithmetic operators

Addition (`+`), Subtraction (`-`), Multiplication (`*`), and Division (`/`)

```
    (+ 11 12 13)  ⇒ 36
    (/ 720 5 4 3 2 1)⇒ 6
```

These operators can operate on two or more values as a single step.

Negation:

```
    (- 144)  ⇒ -144
    (- -21)  => 21
```

But not:

    (+ 72) => ~~72~~ **ERROR**

`max` and `min`

```
(max 4 8 -9 3) ⇒ 8
(min 4 8 -9 3) ⇒ -9
```

Like addition, subtraction, multiplication, and division these operate on any number of values, including one value:

```
(max 9) ⇒ 9
(min 9) ⇒ 9
```

L01.1 Exact numbers

Natural numbers (ℕ)

Natural numbers are important in computer science (and especially in CS135)

Natural numbers are defined as follows:

- 0 is a natural number
- If *n* is a natural number, then *n* + 1 is a natural number

Some definitions of natural numbers start at 1, but 0 is more convenient in computer science contexts.

Since ℕ is not on your keyboard, and "natural number" is a lot of typing, we will write "**`Nat`**" to indicate natural numbers when we want to talk about them in Racket.

Integers (ℤ)

Natural numbers are great if we just want to add things together or to subtract small numbers from big numbers, but as soon as we write 1 - 2 = ? we discover we need negative numbers as well.

…, -3, -2, -1, 0, 1, 2, 3, …

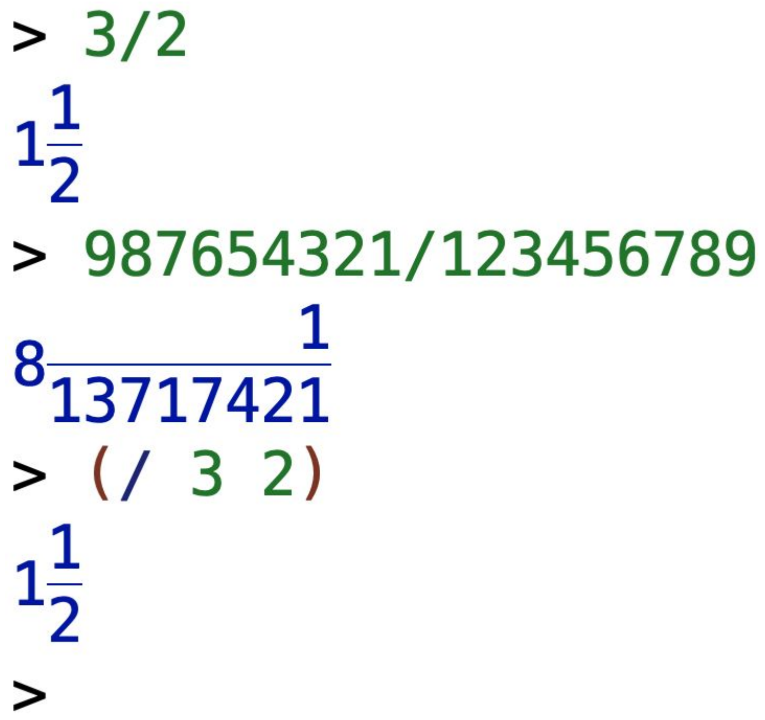
Since ℤ is not on your keyboard we will write "`Int`" to indicate integers.

Rational numbers (ℚ)

Integers are great if we just want to add, subtract, multiple, and divide by factors, but as soon as we write 3/2 = ? we discover we need rational numbers as well.

Since ℚ is not on your keyboard we will write “`Rat`” to indicate rational numbers.

Rational numbers are exact in Racket

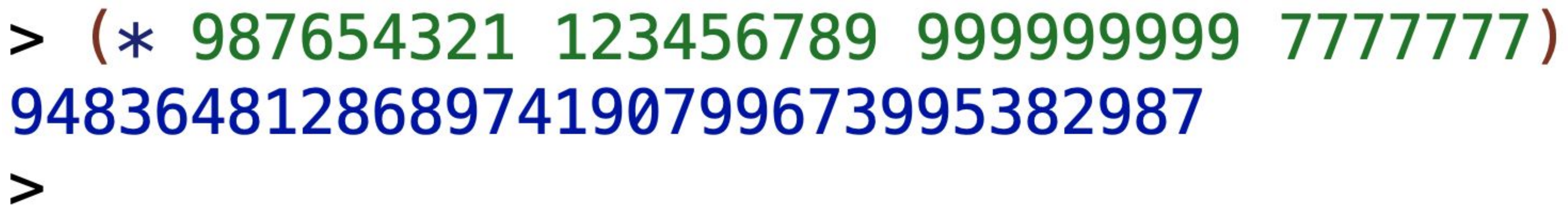


```
>  3/2
 1
1─
 2
>  987654321/123456789
          1
8─────────
 13717421
>  (/ 3 2)
 1
1─
 2
>
```

Note that (/ 3 2) is division, while 3/2 is just a way of expressing the number one-and-a-half.

Try these in DrRacket, if you get results that use decimal notation, you don’t have correct language settings. Go back to Lecture 00 to see how to set them correctly.

`Nat`, `Int` and `Rat` values can be arbitrarily large or small



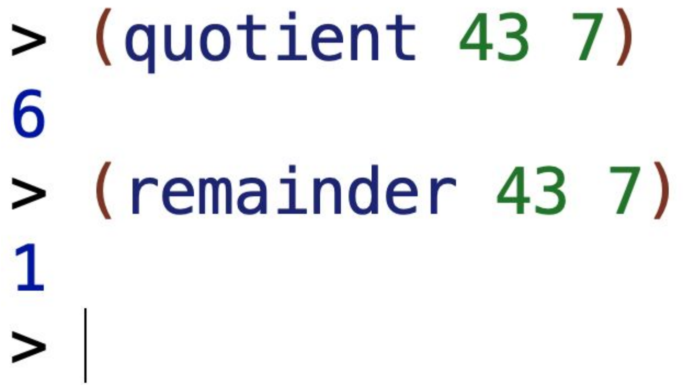
```
> (* 987654321 123456789 999999999 7777777)
948364812868974190799673995382987
>
```

Some programming languages may limit the size of an integer to a range that is related to how physical computers are organized (at least without using a special package or library).

In Racket exact numbers can be as large or small as available memory allows.

We assume our computer is finite, but we can always add memory if we don’t have enough, hence “arbitrarily” and not “infinitely”.

`quotient` and **`remainder`** operations



```
> (quotient 43 7)
6
> (remainder 43 7)
1
> 
```

These operations provide **`Int`** division without creating a **`Rat`**.

They only work on **`Int`**s (and therefore **`Nat`**s) and will produce an error if applied to **`Rat`**s.

We say that **`quotient`** (or **`remainder`**) “consumes” two **`Int`**s and “produces” an **`Int`**.

Types and subtypes

`Nat`, **`Int`** and **`Rat`** are *types* of numbers.

Some operations (e.g., **`remainder`**) are only valid on certain types of numbers.

Any **`Nat`** is also an **`Int`**. Any **`Int`** or **`Nat`** is also a **`Rat`**, we say:

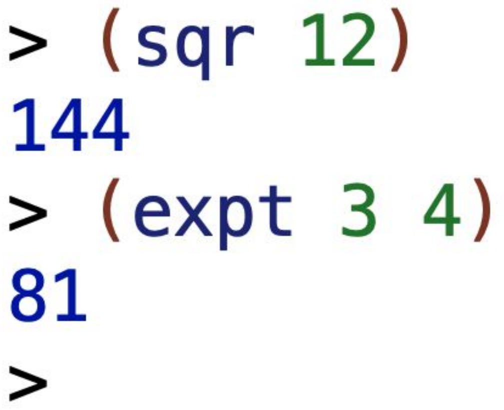
- **`Nat`** is a *subtype* of **`Int`**, and
- **`Nat`** and **`Int`** are *subtypes* of **`Rat`**.

We can also write this relationship as: **`Nat`** $\subseteq$ **`Int`** $\subseteq$ **`Rat`**

If an operator or function works for a type it will automatically work for its subtypes.

The notion of a subtype is important in Computer Science and we will return to it several times during the course.

`sqr` and `expt` operations



```
> (sqr 12)
144
> (expt 3 4)
81
>
```

Exponentiation operators: x^2 and x^y

$12^2 = 144$

$3^4 = 81$

Real numbers ($\mathbb{R}$)

Rational numbers are great if we just want to add, subtract, multiple, divide, and use integer exponents, but as soon as we write $2^{½} = \sqrt{2} = ?$ we discover we need irrational numbers as well, i.e. we need real numbers.

Unfortunately, irrational numbers can't be represented exactly in finite memory.

We'll come back to this problem in Lecture 02, when we discuss inexact numbers.

L01.2 Constant definitions

~~Variables~~ Constants

In a math class we might write:

$$\begin{aligned} x &= 7 \\ y &= 11 \\ x^2 + 6xy + y^2 + 9x - 3y - 100 &= ? \end{aligned}$$

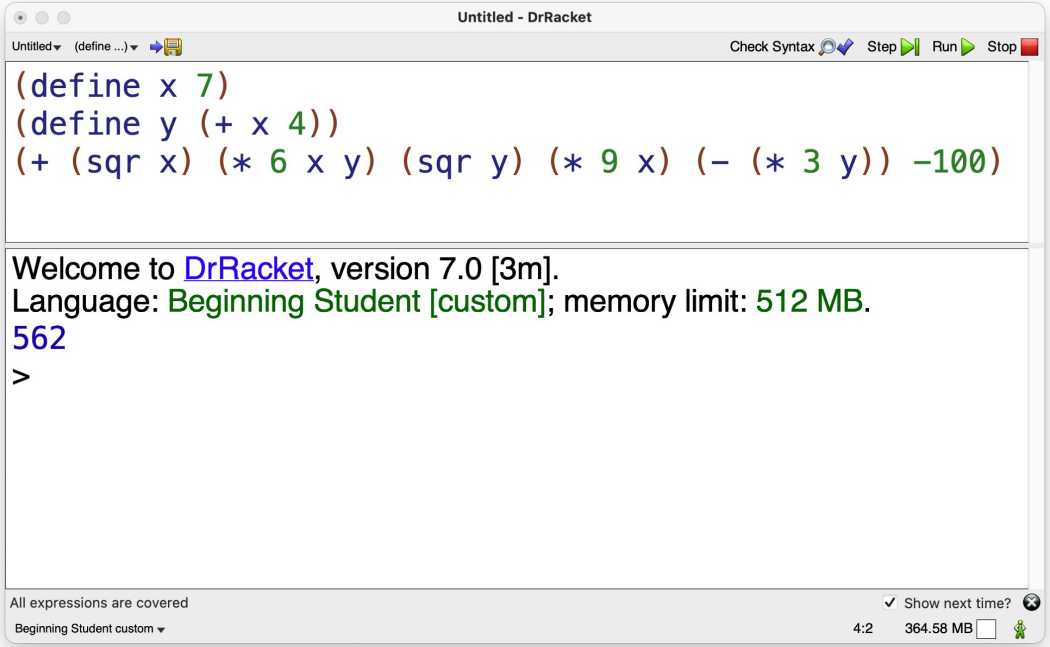
In a math class *x* and *y* are called “variables”, but in racket we call them “constants” because they are defined once and never change.

In Racket we write:

```
(define x 7)

(define y 11)
```

We say that **`define`** “binds” a value to a name



```
(define x 7)
(define y (+ x 4))
(+ (sqr x) (* 6 x y) (sqr y) (* 9 x) (- (* 3 y)) -100)
```

```
Welcome to DrRacket, version 7.0 [3m].
Language: Beginning Student [custom]; memory limit: 512 MB.
562
>
```

All expressions are covered ✓ Show next time?

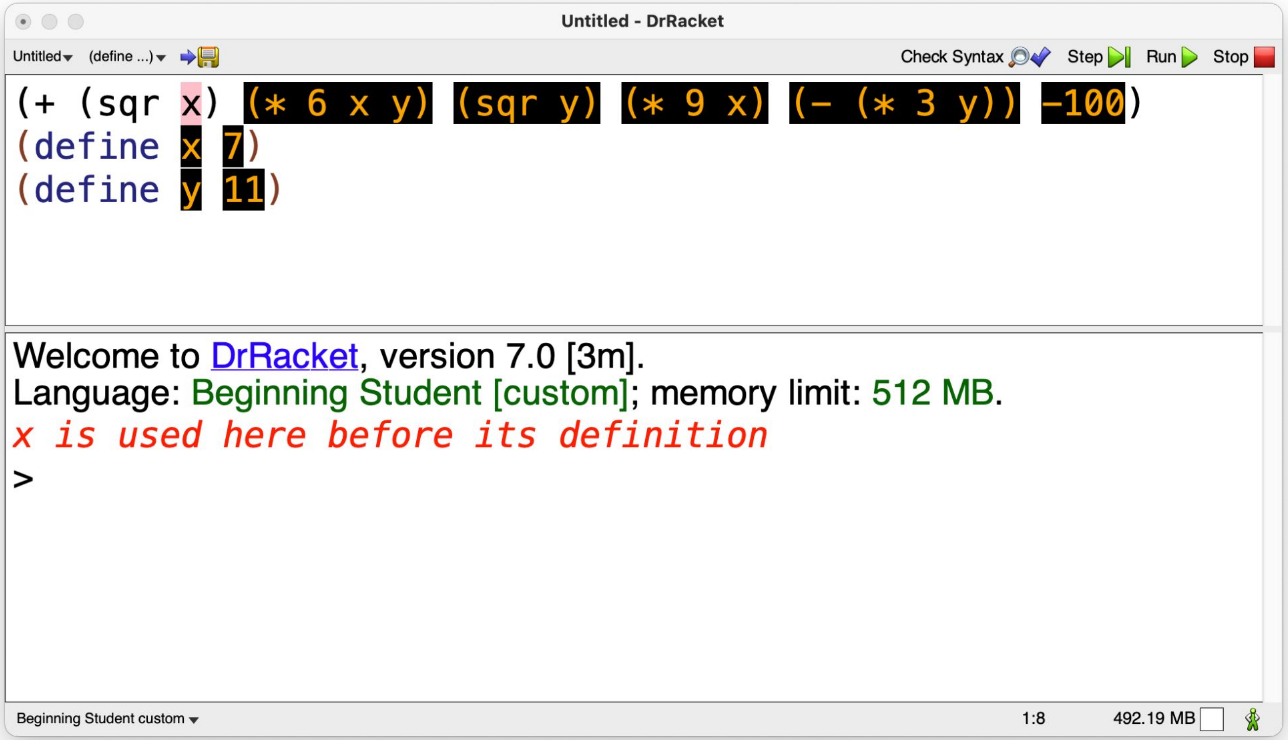
Beginning Student custom 4:2 364.58 MB

`(define x 7)`
binds the value `7` to the name **`x`**

`(define y (+ X 4))`
is evaluated in two substitution steps:

1. **`(+ X 4)`** is evaluated to give `11`.
2. **`y`** is bound to that value.

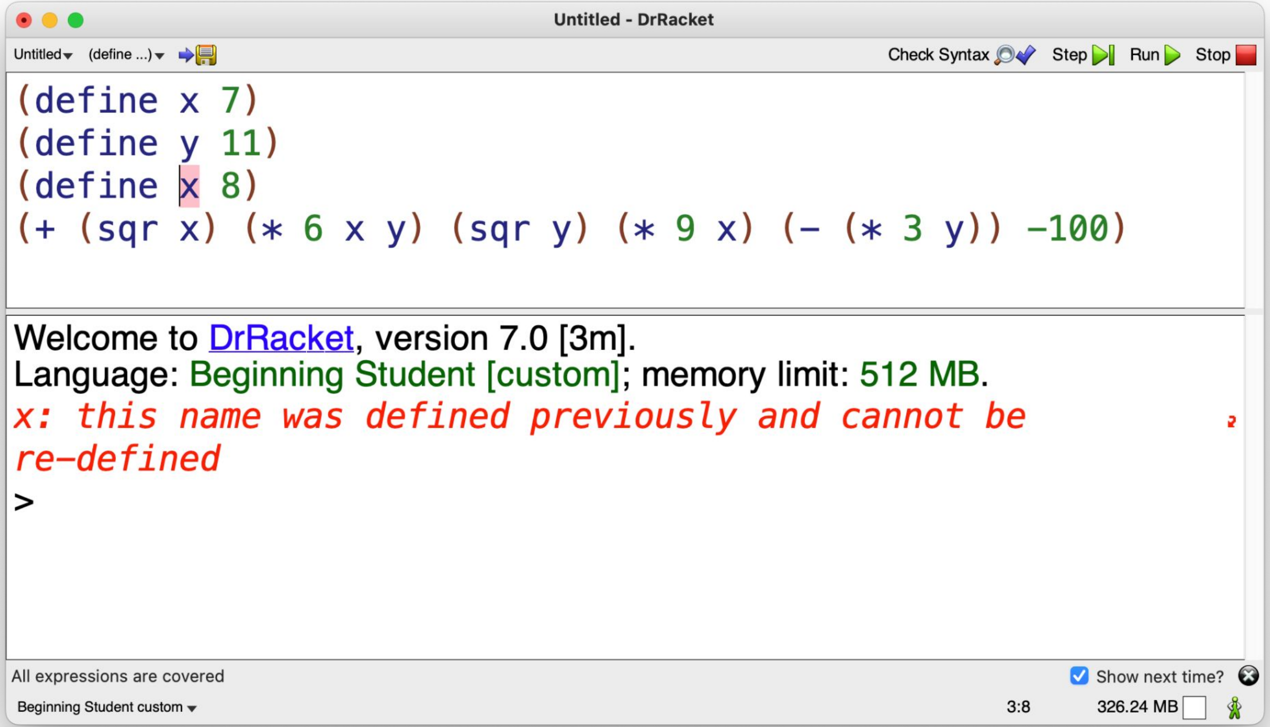
Names must be defined before they are used



```
(+ (sqr x) (* 6 x y) (sqr y) (* 9 x) (- (* 3 y)) -100)
(define x 7)
(define y 11)
```

```
Welcome to DrRacket, version 7.0 [3m].
Language: Beginning Student [custom]; memory limit: 512 MB.
x is used here before its definition
>
```

Once defined, names can't be re-defined.



```
(define x 7)
(define y 11)
(define x 8)
(+ (sqr x) (* 6 x y) (sqr y) (* 9 x) (- (* 3 y)) -100)
```

```
Welcome to DrRacket, version 7.0 [3m].
Language: Beginning Student [custom]; memory limit: 512 MB.
x: this name was defined previously and cannot be
re-defined
>
```

L01.3 Boolean expressions

What does “<” mean?

Consider the expression “x < 5”.

In a math class, it tells us something about x:
    Whatever value x has, that value is less than 5.

We might combine the statement “x < 5” with the statements, “x is a natural number”, “x is even” and “x is a perfect square” to conclude “x is 4”.

In Racket, “**`<`**” means something different, since a constant such as **`x`** already has a value. In Racket “**`<`**” is an operator, like “**`+`**” or “**`sqr`**”.

What does “<” mean?

Suppose we define a constant:

```
(define x 2)
```

Using prefix notation we create a Racket expression equivalent to “x < 5”:

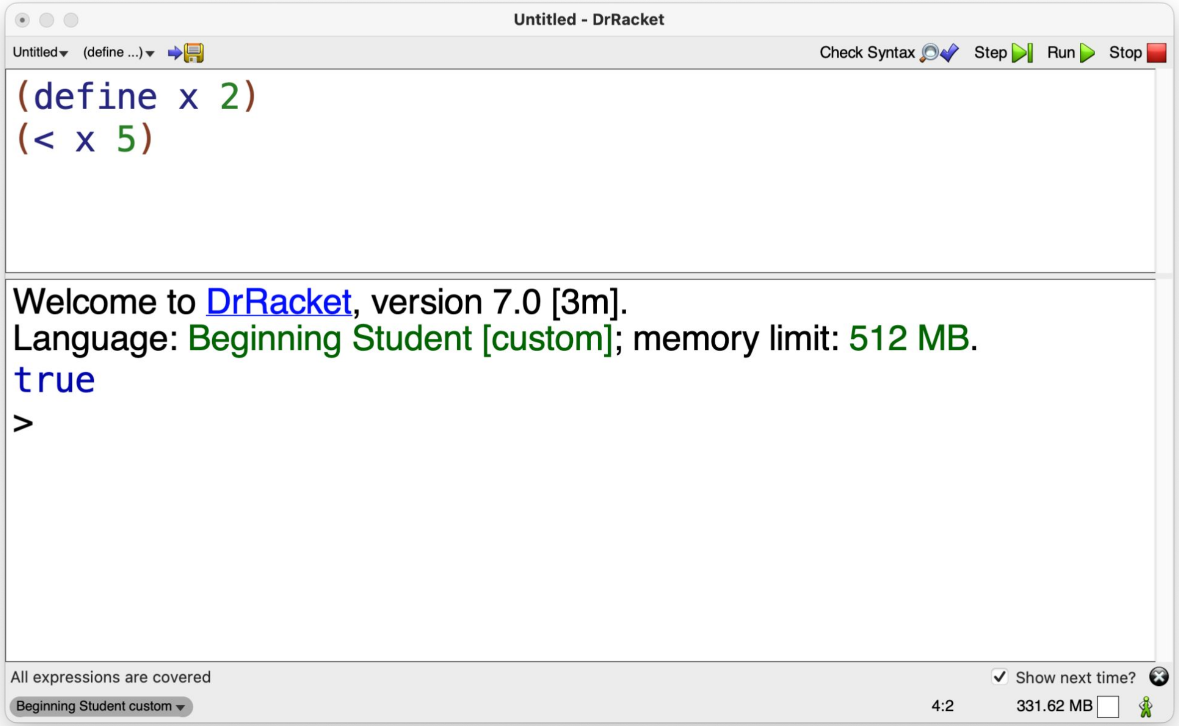
```
(< x 5)
```

This is asking “Is it true that the value of x is less than 5?”

If we evaluate `(< x 5)`, we substitute in the value of the constant, so our expression becomes `(< 2 5)`.

Since it is true that 2 < 5, the statement evaluates to `true`.

What does “<” mean?



```
(define x 2)
(< x 5)
```

```
Welcome to DrRacket, version 7.0 [3m].
Language: Beginning Student [custom]; memory limit: 512 MB.
true
>
```

If you get **`#true`** or something other than **`true`**, you don’t have the correct language settings. Follow the instructions in Lecture 00 to set them correctly.

Boolean values (`Bool`)

The operators <, >, <=, >=, and = compare two numbers and produce a “Boolean” value, a `Bool`.

```
(define x 4)
    (< x 6)      ⇒ is x less than 6?
    (> x 6)      ⇒ is x greater than 6?
    (= x 7)      ⇒ is x equal to 7?
    (>= 5 x)     ⇒ is 5 greater than or equal to x?
    (<= 5 x)     ⇒ is 5 less than or equal to x?
```

Each evaluates to `true` or `false`. These are the only possible values for a `Bool`.

Expressions with Boolean values

We combine Boolean values using the operators **`and`**, **`or`**, and **`not`**.

- **`and`** has value **`true`** when all of its arguments have value **`true`**; **`false`** otherwise.
- **`or`** has value **`false`** when all of its argument have value **`false`**; **`true`** otherwise.
- **not** has value **`true`** if its argument is **`false`**; **`false`** if its argument is **`true`**.

Both **`or`** and **`and`** require at least two arguments, but may have more.

```
(and (or (< 2 3) (> 1 5)) (not (< 10 15)) (<= 5 5))
⇒ (and (or true (> 1 5)) (not (< 10 15)) (<= 5 5))
⇒ (and true (not (< 10 15)) (<= 5 5))
⇒ (and true (not true) (<= 5 5))
⇒ (and true false (<= 5 5))
⇒ false
```

Short-circuit evaluation

We could equivalently define **`and`** and **`or`** as:

- **`and`** has value **`false`** if any of its arguments have value **`false`**; **`true`** otherwise.
- **`or`** has value **`true`** if any of its arguments have value **`true`**; **`false`** otherwise.

Under this definition we may not need to fully evaluate all subexpressions:

```
(or (< 2 3) (> 1 5)) ⇒ (or true (> 1 5)) ⇒ true

(and true (not true) (<= 5 5)) ⇒ (and true false (<= 5 5))⇒ false
```

Exploiting this definition to reduce the number of substitution steps is called “short-circuit evaluation”.

Lecture 01 Summary

What happens next?

Over four lectures we will develop our model of computation:

1. **Values and expressions**
2. Functions
3. Conditional expressions
4. Recursion

After the final step, we will have built a complete “computer”, essentially from math.

We will then add “lists” to our model of computation to simplify data organization.

We will then explore a variety of basic algorithms and data structures using lists.

L01: You should know

- Exact numbers: `Nat`, `Int`, and `Rat.` Subtype relationships.
- Writing Racket mathematical expressions in prefix notation
- Applying substitution steps to evaluate Racket expressions
- Defining Racket constants with `define`
- Operators: `+`, `-`, `*`, `/`, `quotient`, `remainder`, `sqr`, `expt`
- Operators: `max` and `min`
- Operators: <, >, =, <=, >=
- `Bool` expressions with `true`, `false`, `and`, `or`, `not`

L01: Allowed constructs

`( ) + - * / = < > <= >= and define expt false max min not or quotient remainder sqr true Bool Int Nat Rat`

Each lecture ends with a list of "allowed constructs", which grows over the term.

The front page of each assignment indicates the allowed constructs for that assignment. For example "Allowed constructs: L09" *meas that you can use any construct listed for Lecture 9. Other restrictions may also be listed on the first page of an assignment, so make sure you read the entire first page before starting.*

If you spot a missing construct, let us know on Piazza, but the answer to the question, "Can I use …?" is normally "Is it on the list of Allowed Constructs".