$\boxed{2}$ Pure LISP, Arithmetic, and Rewrite Rules

$\boxed{2.1}$ Numbers and Arithmetic Operators

Integers: `1` `9` `-55` `12345678901234567890`

Floating Point Numbers: `1.0` `0.1` `-0.09` `3e6`

Rationals: `1/3` `57/64` `-2/3` `1234567890987654321/13`

Arithmetic Operators: `+` `-` `*` `/`

`(+ 9 5)` $\Longrightarrow$ `14`

`(- 9 5)` $\Longrightarrow$ `4`

`(* 9 5)` $\Longrightarrow$ `45`

`(/ 9.0 5.0)` $\Longrightarrow$ `1.8`

Expression Syntax: Prefix, Surrounded by Parentheses,
<u>No</u> Commas

`(+ (* 9 5) (- 9 5))` $\Longrightarrow$ `(+ 45 4)` $\Longrightarrow$ `49`

2.2 Exact Arithmetic and Special Cases

`(* 2/3 1/2)` $\Longrightarrow$ `1/3`   `(/ 2 3)` $\Longrightarrow$ `2/3`

`(* 123456789 987654321)` $\Longrightarrow$ `121932631112635269`

`(/ 2.0 3.0)` $\Longrightarrow$ `0.6666666666666666`   [inexact]

Divide and Round Down (toward $-\infty$) to Integer:

`(FLOOR 17 5)` $\Rightarrow$ `3`   `(MOD 17 5)` $\Rightarrow$ `2`   `17 = 3 * 5 + 2`

`(FLOOR -17 5)` $\Rightarrow$ `-4`   `(MOD -17 5)` $\Rightarrow$ `3`   `17 = -4 * 5 + 3`

Special Cases: `(+)` $\Longrightarrow$ `0`   `(*)` $\Longrightarrow$ `1`   `(* 5)` $\Longrightarrow$ `5`

`(+ 1 2 3 4 5)` $\Longrightarrow$ `15`   `(* 1 2 3 4 5)` $\Longrightarrow$ `120`

Comments:

1. The reason for the unusual syntax of LISP will become clear in the next few lectures.

2. The reason LISP has arbitrarily large integers and rational numbers is that LISP was originally designed for writing programs that did computations symbolically, with variables and equations, rather than just numerically. This will also become clearer later.

3. The LISP `+` operator can add zero or more numbers. If asked to add zero numbers, it returns `0`.

4. Similarly the LISP `*` operator can multiply zero or more numbers. If asked to multiply zero numbers, it returns `1`.

5. Since `(/ 10 3)` returns `10/3` and not `3`, one must use another function if one wanted to get `3`. That function is `floor`, which when used with two positive integers, returns the integer part of their quotient.

6. When you divide `-17` by `5` in America, you get `-3` with remainder `-2`. In some other countries (Europe?) you get `-4` with remainder `3`. Americans round the factional quotient toward zero, and in other countries people round it toward minus infinity. Mathematicians prefer to round toward minus infinity, because then the remainder is better behaved in most mathematical arguments. The remainder for the round-toward-minus-infinity division is called the “modulo” function: we say `-17` modulo `5` is `3`.

7. In LISP arithmetic with floating point numbers yields floating point numbers of some precision, e.g. 16 decimal digits, which are approximate answers, unlike the exact answers of rational arithmetic.

$\boxed{2.3}$ Rewrite Rules and Function Definitions

Rewrite Rule (RR):

$\mathtt{n} \in$ number `(CIRCLE-AREA n n)` $\Longrightarrow$ `(* 3.1415 n n)`

'is a' 'rewrites to'

LISP: `(DEFUN CIRCLE-AREA (N) (* 3.1415 N N))`

C: `float circle_area (float n) { return 3.1415 * n * n; }`

RR: `(SQUARE n)` $\Longrightarrow$ `(* n n)` $\mathtt{n} \in$ number

Example: `(SQUARE 1/3)` $\Longrightarrow$ `(* 1/3 1/3)` $\Longrightarrow$ `1/9`

LISP: `(DEFUN SQUARE (N) (* N N))`

C: `float square (float n) { return n * n; }`

2.4 True/False, Relations, Predicates

Special Values: `NIL` `T` ‘::=’ means ‘is a’ syntactically

false ::= `NIL`

true ::= any-non-`NIL`-value [default `T`]

Relations on Numbers: `<` `>` `=` `/=` `<=` `>=`

`(< 5 6)` $\Longrightarrow$ `T`	`(> 5 6)` $\Longrightarrow$ `NIL`	`(= 5 6)` $\Longrightarrow$ `NIL`
`(<= 5 6)` $\Longrightarrow$ `T`	`(>= 5 6)` $\Longrightarrow$ `NIL`	`(/= 5 6)` $\Longrightarrow$ `T`

`(< 5 6 7 8)` $\Longrightarrow$ `T`	`(= 5 5 5 5)` $\Longrightarrow$ `T`
`(< 5 9 7 8)` $\Longrightarrow$ `NIL`	`(= 5 5 6 6)` $\Longrightarrow$ `NIL`

Predicates: `(EVENP i)` `(ODDP i)` `i` $\in$ integer

$\boxed{2.5}$ Conditional Expressions, MAX

test → ← value

```
(COND (e1 v1)
      (e2 v2)
      . . .   )
```

$\Longrightarrow$ the first $\mathtt{v}_i$ such that $\mathtt{e}_i$ is true

RR: $\mathtt{(MAX\ m\ n)} \Longrightarrow \mathtt{m}$ if $\mathtt{m} \geq \mathtt{n}$ $\qquad$ $\mathtt{m}, \mathtt{n} \in$ number

$\Longrightarrow \mathtt{n}$ otherwise

LISP:

```
(DEFUN MAX (M N)
  (COND ((>= M N)  M)
        (T         N)
```

$\boxed{2.6}$ Rewrite Rules, COND

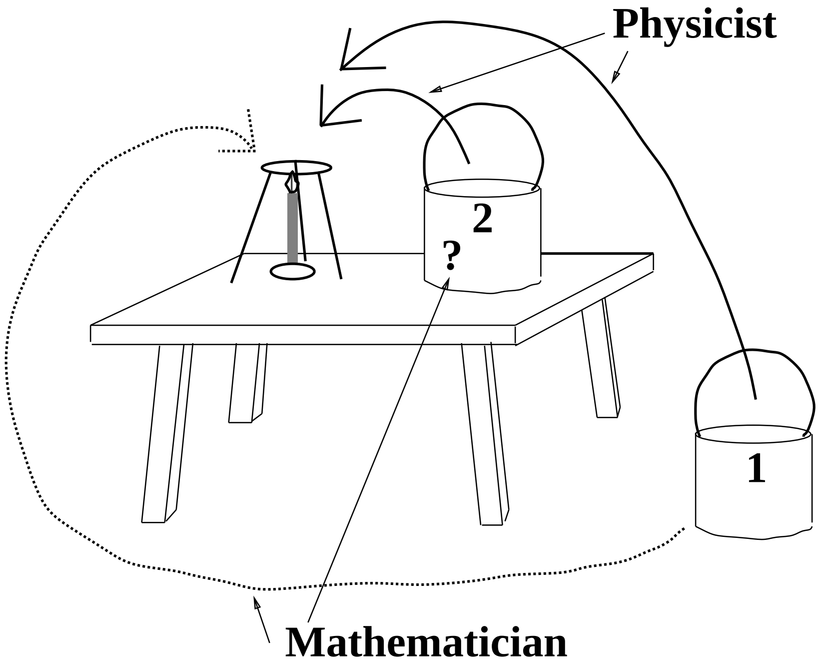
RR means: Rewrite a piece of code as another
piece of code that does the <u>same</u> thing

```
(COND (NIL v1)   ==>   (COND (e2 v2)
      (e2  v2)                . . .  )
      . . .   )

(COND (T  v1)   ==>   v1
      . . .   )


(MAX 2 9.7) ==> (COND ((>= 2 9.7) 2)  ==> (COND (NIL 2)
                      (T       9.7))             (T 9.7)
            ==> (COND (T 9.7)) ==> 9.7
```

$\boxed{2.7}$ Recursion



Recursion: Reduce a problem to the <u>same</u> problem
but with <u>smaller</u> input

Smallest Input: Base Case

$\boxed{2.8}$ FACTORIAL

Example: $5! \Longrightarrow 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \Longrightarrow 120$

RR: $0! \Longrightarrow 1$

$n! \Longrightarrow n \cdot (n-1)!$ if $n > 0$, $n \in$ integer

LISP RR: `(FACTORIAL 0)` $\Longrightarrow$ `1`

`(FACTORIAL n)` $\Longrightarrow$ `(* n (FACTORIAL (- n 1)))`

`n` $\in$ integer if `(> n 0)`

LISP:

```
(DEFUN FACTORIAL (N)                ;Assumes N >= 0.
  (COND ((= N 0) 1)
        (T (* N (FACTORIAL (- N 1))))))
```

$\boxed{2.9}$ Rewrite Rules Clarified

The rule: `(FACTORIAL n)` $\Longrightarrow$ `(* n (FACTORIAL (- n 1)))`

`n` $\in$ integer if `(> n 0)`

is actually a <u>meta</u>-rule that denotes an infinite number of ordinary rewrite rules:

```
(FACTORIAL 1) ==> (* 1 (FACTORIAL (- 1 1)))
(FACTORIAL 2) ==> (* 2 (FACTORIAL (- 2 1)))
(FACTORIAL 3) ==> (* 3 (FACTORIAL (- 3 1)))
```

etc. for `n` = 1, 2, 3, 4, 5, . . .

`n` is a <u>meta</u>-variable, and <u>not</u> a LISP variable.
We often drop the 'meta' from 'meta-rule',
but <u>not</u> from 'meta-variable'.

Comments:

8. We put meta variables in lower case, and LISP symbols in upper case, in these view-graphs. LISP is actually case insensitive: a LISP variable or function name can be written in upper or lower case and will mean the same thing (not like C).

9. You can tell whether something is a meta-rule by looking to see whether it contains a meta-variable. Therefore we drop the 'meta' from meta-rule, and just talk about a 'rewrite rule', when what we really mean is a 'rewrite meta-rule'.

10. We distinguish meta-variables by giving each a range, such as '$n \in$ integer'. In text files we write '`n is an integer`' instead, because we cannot use the '$\in$' symbol. Meta-variables are just those variables with ranges given in these ways.

Comments:

11. When we write code or rewrite rules in code we tend to put everything in lower case, just because it is easier to type.

12. The general format of a rewrite (meta-)rule in code is:

```
meta-expression-1 ===> meta-expression-2
                                conditions
ranges
```

    A meta-expression is just an expression which becomes an expression of the programming language (e.g. LISP) after values are substituted for all the meta-variables. An example of a rewrite meta-rule written in code (with lower case) is:

```
(factorial n) ===> (* n (factorial (- n 1)))
                                if n >= 1
n is an integer
```

    where '`if n >= 1`' is a condition.

13. Rewrite rules are used extensively in this course, and students should master them as soon as possible.

14. Ditto recursion.

2.10 Evaluation by Rewrites and Calls

```
(FACTORIAL 2) ==> (COND ((= 2 0) 1)
    |                   (T (* 2 (FACTORIAL (- 2 1)))))
    |         ==> (COND (NIL 1)
call |                  (T (* 2 (FACTORIAL (- 2 1)))))
    |         ==> (* 2 (FACTORIAL (- 2 1)))
    ↓         ==> (* 2 (FACTORIAL 1)) ==> (* 2 1) ==> 2
(FACTORIAL 1) ==> ... ==> 1  <-┘  call
    ↓ call
(FACTORIAL 0) ==> ... ==> 1
```

$\boxed{2.11}$ POWER

Examples: `(POWER 2 3)` $\Longrightarrow$ 8 i.e. 2^3

`(POWER 3 2)` $\Longrightarrow$ 9 i.e. 3^2

`(POWER m n)` $\Longrightarrow$ m^n

RR: `(POWER m 0)` $\Longrightarrow$ 1

`(POWER m n)` $\Longrightarrow$ `(* m (POWER m (- n 1)))` if `n > 0`

`m` $\in$ number, `n` $\in$ integer

$\boxed{2.12}$ Call Trees, Time Complexity

```
(FACTORIAL 2)                (POWER 0.3 2)      <-- root
    ↓ call                       ↓ call
(FACTORIAL 1)                (POWER 0.3 1)
    ↓ call                       ↓ call
(FACTORIAL 0)                (POWER 0.3 0)      <-- leaf
```

Call Tree for `(FACTORIAL n)` has $1 + n \;=\; O(n)$ nodes.
Call Tree for `(POWER m n)` has $1 + n \;=\; O(n)$ nodes.

In Computer Science (but not math) "$= \; O(n)$" means:
"$\leq \; C_1 + C_2 \cdot n$ for some constants C_1, C_2"

$\boxed{2.13}$ FAST-POWER, Time Complexity

`POWER` RR

$m^0 \Longrightarrow 1$

$m^n \Longrightarrow m \cdot m^{n-1}$ if $n > 0$

`FAST-POWER` RR

$m^0 \Longrightarrow 1$

$m^n \Longrightarrow (m^{\frac{n}{2}})^2$ if $n > 0$ even

$m^n \Longrightarrow m \cdot (m^{\frac{n-1}{2}})^2$ if n odd

(FAST-POWER 0.3 7)

(FAST-POWER 0.3 3) (SQUARE 0.3^3)

(FAST-POWER 0.3 1) (SQUARE 0.3^1)

(FAST-POWER 0.3 0) (SQUARE 0.3^0)

Tree Size $\leq 3 + 2 \cdot \log_2 n \;=\; O(\log n)$ nodes

i.e.: $\leq C_1 + C_2 \cdot \log n$ for $C_1 = 3$ and $C_2 = \frac{2}{\log 2}$

Time Complexity of `FAST-POWER` $= O(\log n)$

Comments:

15. Every program execution defines a call tree, whose nodes are the function executions of the program, labeled by the name of the function and its arguments. An arrow is drawn from one function execution to a second function execution if the first executed the call that started the second.

16. The size of a program execution's call tree is an indicator of the execution time of the program. If the program does not have any long loops inside a function execution, the call tree size is a good indicator.

17. The behavior of the execution time of a program given an input of size n as n becomes large is called the time complexity of the program. For example, the time complexity of `FACTORIAL` is $O(n)$ and the time complexity of `FAST-POWER` is $O(\log n)$.

18. Our definition of "$= O(n)$" is different from but equivalent to the mathematicians definition in the cases we are interested in. We prefer our definition because it can be applied more directly.

Comments:

19. The mathematicians definition of "$= O(n)$" is: $\leq C_1 \cdot n$ for all $n > C_2$, for some constants C_1 and C_2.

    This definition is equivalent to our definition: $\leq C_1 + C_2 \cdot n$ for all $n \geq 0$, for some constants C_1 and C_2.

20. However, the mathematicians definition of "$= O(\frac{1}{n})$" would be: $\leq C_1 \cdot \frac{1}{n}$ for all $n > C_2$, for some constants C_1 and C_2.

    This is <u>not</u> equivalent to $\leq C_1 + C_2 \cdot \frac{1}{n}$ for all $n \geq 1$, for some constants C_1 and C_2 (we skip the case $n = 0$ to avoid problems with $\frac{1}{0}$).

    In general, for $= O(f(n))$ where $f(n)$ is some function of n, our definition and the mathematician's agree if the function $f(n)$ stays above some positive constant as n goes to positive infinity. This will be true when measuring the size of computations, since in these cases $f(n)$ will be a positive increasing function of n.