CSE1729: Introduction to Programming

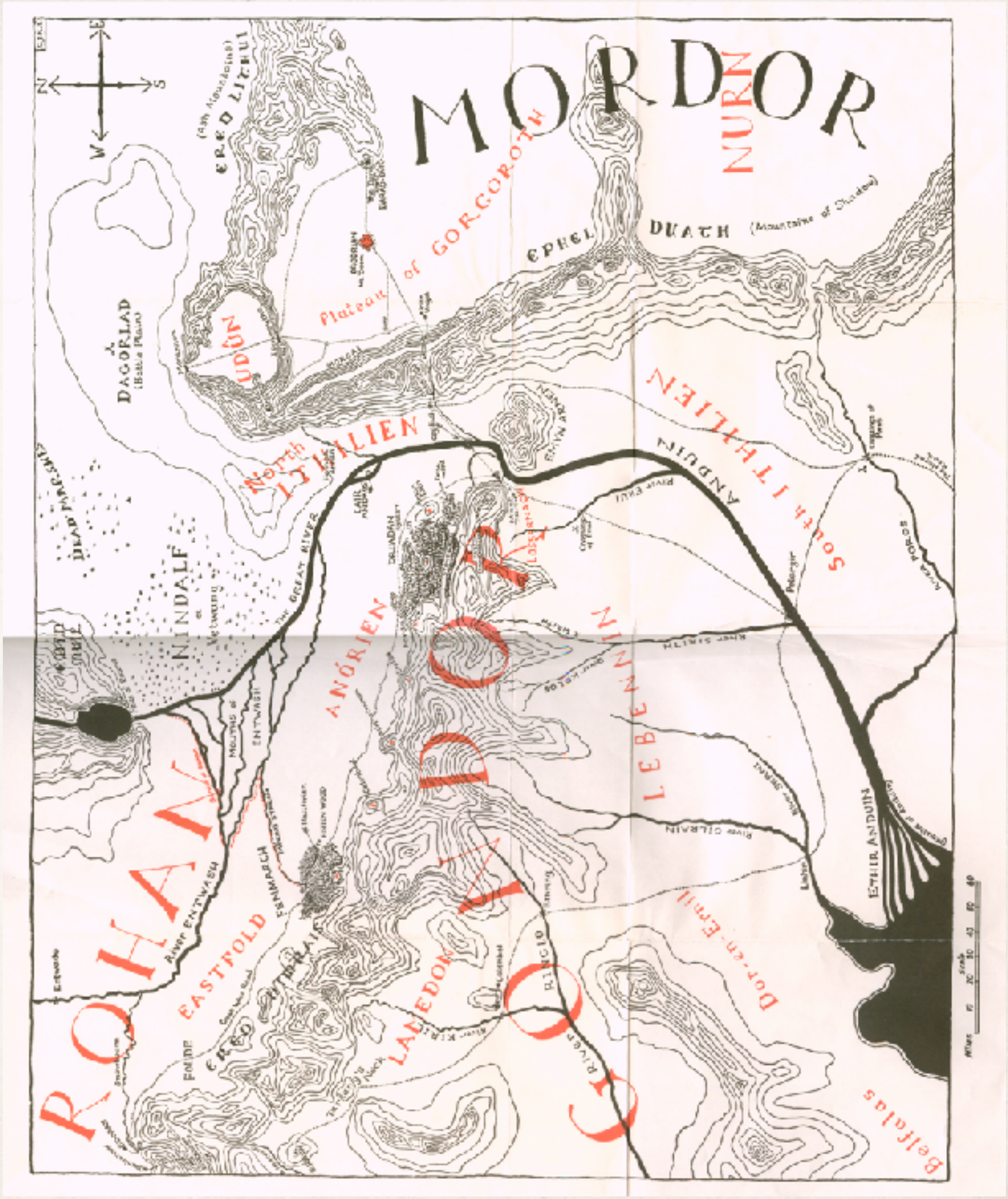
Elements of SCHEME programming

Greg Johnson

Course Survey

Computing is one of mankind's greatest discoveries

This is your path...



...to programming wizardry.

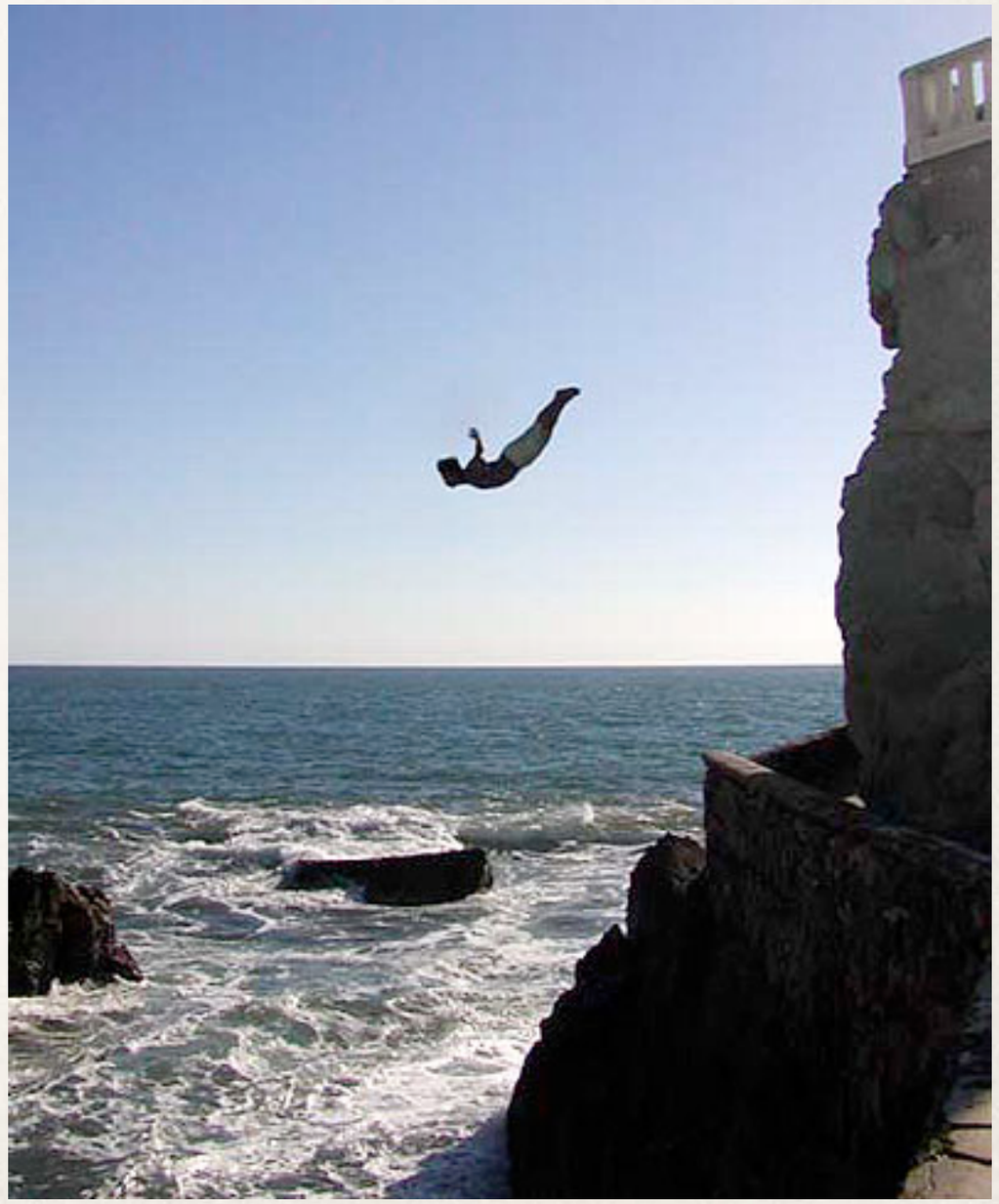
```
public class Quine
{
  public static void main( String[] args )
  {
    char q = 34;      // Quotation mark character
    String[] l = {    // Array of source code
    "public class Quine",
    "{",
    "  public static void main( String[] args )",
    "  {",
    "    char q = 34;      // Quotation mark character",
    "    String[] l = {    // Array of source code",
    "    ",
    "    };",
    "    for( int i = 0; i < 6; i++ )           // Print opening code",
    "        System.out.println( l[i] );",
    "    for( int i = 0; i < l.length; i++ )    // Print string array",
    "        System.out.println( l[6] + q + l[i] + q + ',' );",
    "    for( int i = 7; i < l.length; i++ )    // Print this code",
    "        System.out.println( l[i] );",
    "  }",
    "}",
    };
    for( int i = 0; i < 6; i++ )           // Print opening code
        System.out.println( l[i] );
    for( int i = 0; i < l.length; i++ )    // Print string array
        System.out.println( l[6] + q + l[i] + q + ',' );
    for( int i = 7; i < l.length; i++ )    // Print this code
        System.out.println( l[i] );
  }
}
```

It turns that writing programs for complicated procedures is...

- ...*Complicated*. Over the years, computer scientists have developed a number of powerful tools for managing complexity:
  - data abstraction, abstract data types, object-orientation;
  - functional abstraction, functional programming and meditative mutation, scope control;
  - and...most importantly...programming practices that let the structure of the program represent the structure of the problem.
- Development of these tools will be our principal goals in 1729.

WARNING

- **This will be an unusually demanding and time-consuming class:**
  - Taught at a high level of abstraction
  - Uses a celebrated, but challenging textbook
  - Will attempt to cover, in a single semester, most of elements of modern programming design



Why would anyone take this class?

- Taught in an *extremely simple and beautiful* programming language: SCHEME.

- Focuses on major "structural design" principles: functional and data abstraction. These are the tools that seasoned computer scientists use to reason about building large, complex programs.

- Develops a strong conceptual foundation for more advanced topics in computing. When CSE2050 arrives, you'll already know how to merge heaps and why computer scientists like bushy trees.

- Great investment yields great reward...and...after CSE1729, you'll never be afraid of recursion.

- Staff...

We're here to help you learn

- Office hours on Friday (see course website). If you have questions, we will answer them for a compliment or small fee.

- Labs should be fun and collaborative.

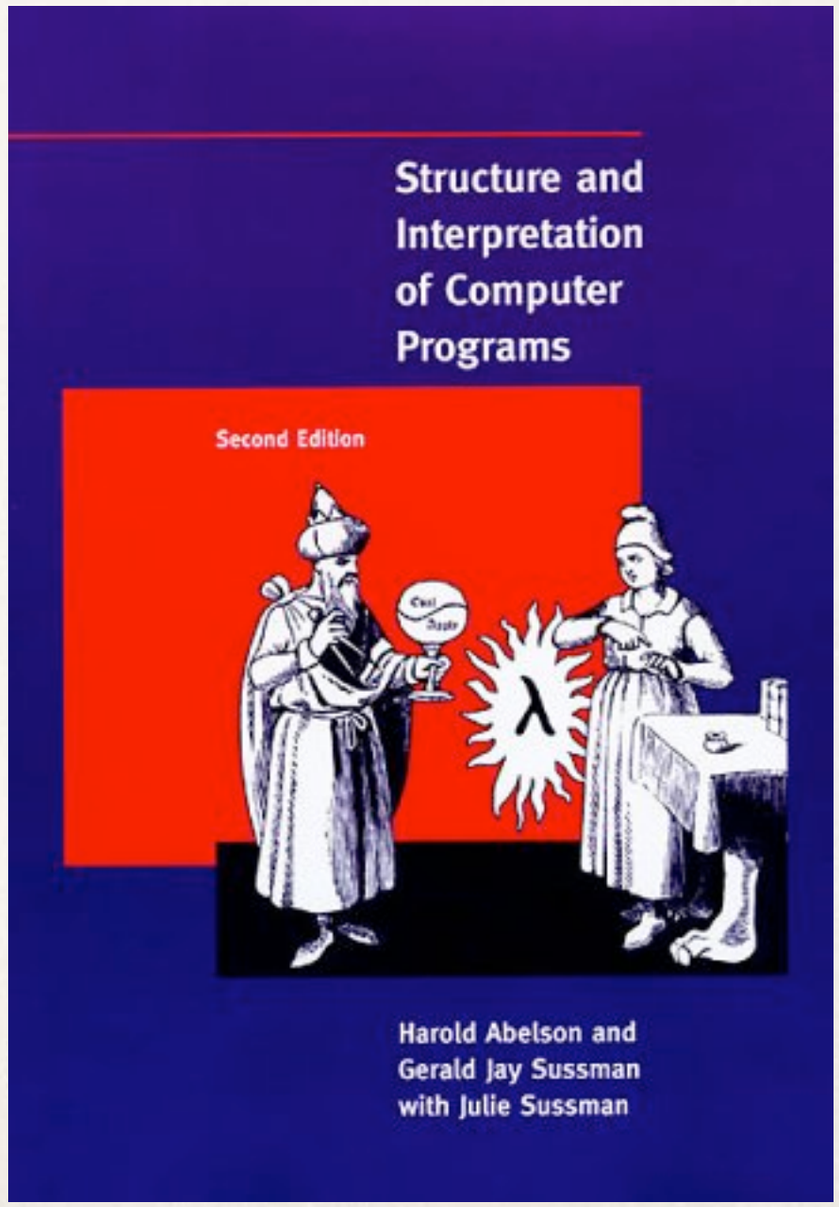
- Technical forum on the Moodle site; use it for your technical questions 24 hours a day. Then others can benefit from the question and the solution.

The textbook

- **Freely available**, on-line, as an html, .pdf, and e-reader document.

- Also available at the Campus store. A more expensive option, but convenient.

- (Of course, we do not insist that you purchase the book.)



Syllabus

- Roughly, the first three chapters of **Structure and Interpretation of Computer Programs**, by Abelson and Sussman:
  - An introduction to the SCHEME interpreter.
  - Basic functional abstraction in a purely functional language. Recursive programming, numeric and list processing, functions as first-class objects, rewrite semantics.
  - State and environment semantics. Lexical scoping. Data abstraction via objects (and possibly streams as time permits)

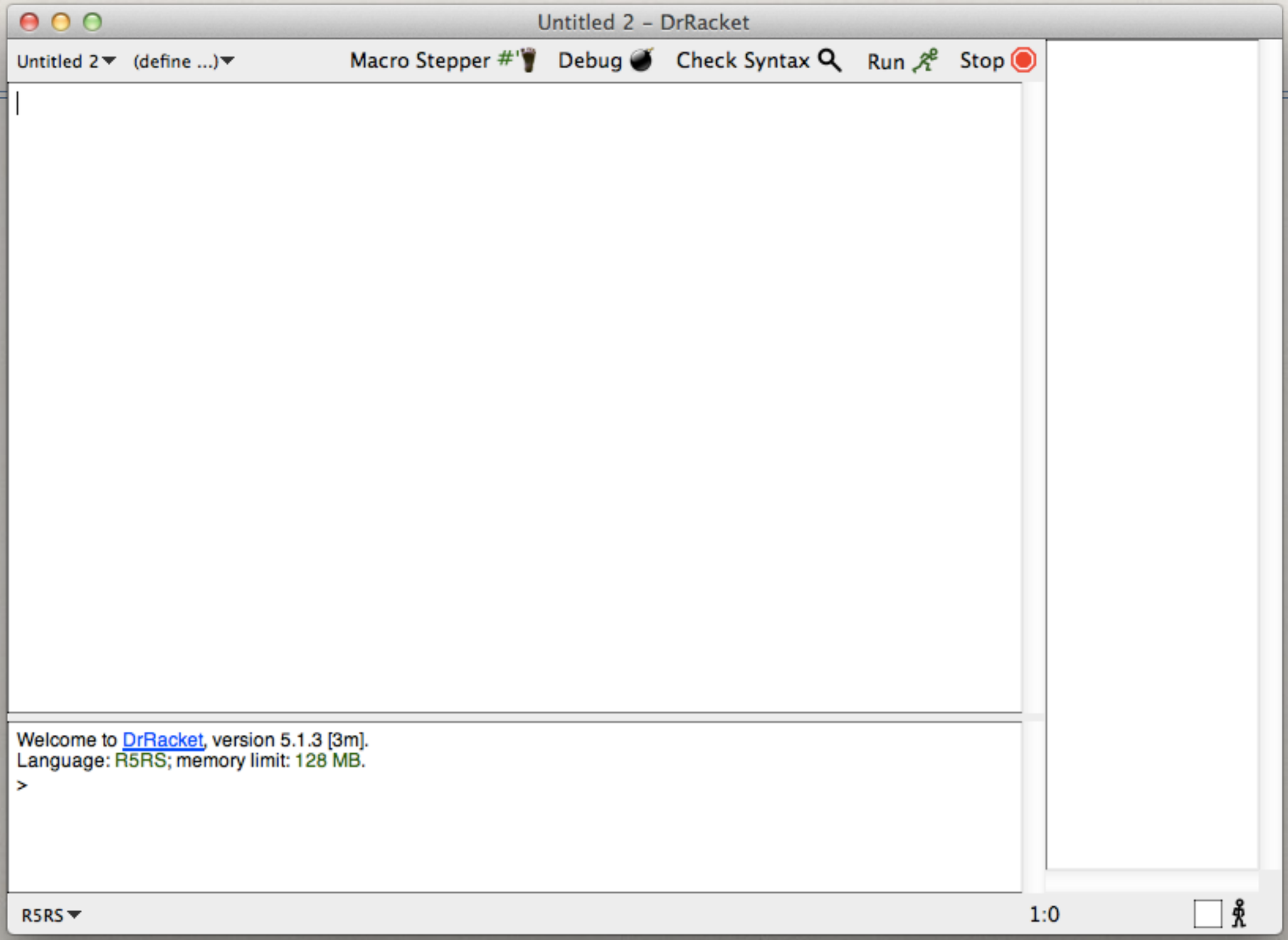
The programming environment

- We will use **Racket**.
- Freely available for Mac, Linux, and even Windows.

  http://racket-lang.org/

- Additionally installs a friendly graphical front-end.
- Installed in the lab. If you have a computer of your own, we recommend installing it immediately to get started!

Racket IDE


Untitled 2 – DrRacket
Untitled 2
(define ...)
Macro Stepper
Debug
Check Syntax
Run
Stop
Welcome to DrRacket, version 5.1.3 [3m].
Language: R5RS; memory limit: 128 MB.
>
R5RS
1:0

Administrative Details

Administration

- Your lecturer: Greg Johnson
  - e-mail: greg.johnson@uconn.edu.
    Office hours: Friday 10am-12pm and by appointment, ITEB 255.
  - Your TA: TBD
- Course administration by a **Moodle** system (assignment submission & grading, course notes, extra material, links to software and textbook, &c.) You should have already received information for setting up an account and enrolling. The website is:

http://courses.engr.uconn.edu/

Course meetings; Teaching assistant

- **Lecture**: Monday & Wednesday, 11:15-12:05, BPB 131.

- **Laboratory sessions**: Tuesdays. These are a required part of the course. Managed by the Teaching Assistants.

- Teaching assistants:

TA	Office	Office Hours
TBD	ITEB 140	TBD

Course Structure: Problem Sets

- **Assigned and due every week** They will be assigned on Fridays and due 10 days later, on Monday evening. (The hard deadline is Tuesday at 4am.)

- **Submission** They will be handed in & graded on the Moodle system. *All SCHEME source code must be submitted in standard text files. Supplementary typeset material is welcome: please use .pdf or MS Word.*

- **Collaboration** *Problem sets are individual work*; however, we have special forums set up on the Moodle site for technical help. Additionally, supporting material for each problem set will be practiced in the lab.

- **Late work** No late work will be accepted. Instead, we will strike the lowest two homework grades from the record. (So, you may completely neglect two assignments with no penalty.)


IT'S A TRAP!

Plagiarism; academic misconduct

- You
- You lly
by posting your questions to the technical forum on the Moodle site so that others can benefit from your question.
- You sion
wit
- We
Un

Minimum penalty

1st strike: ***at least*** a zero + a letter grade drop on the course

2nd strike: an F in the class.

CSE1729 Honor Code

You have received a copy of the **CSE1729 Honor Code** in lab; it is also on the website. You must pledge to uphold the honor code and to report violations to take the class. Hand in your signed pledge by the end of the week.

http://community.uconn.edu/the-student-code-preamble/

Course Structure: Lab Assignments

- **Timetable**: Lab assignments are due on Tuesday evening, the day of the lab. (The hard deadline in Wednesday morning at 4am.) This timetable is meant to encourage you to do the labwork...in lab!

- **Collaboration**: We encourage you to work with your peers in lab to complete the assignment. Your final submitted work must, however, be prepared on your own.

- **Submission**: As with the problem sets: all lab assignments are to be handed in on the moodle site.

- **Late work/lab absences**: As with problem sets: we will strike the two lowest lab assignment grades from your record.

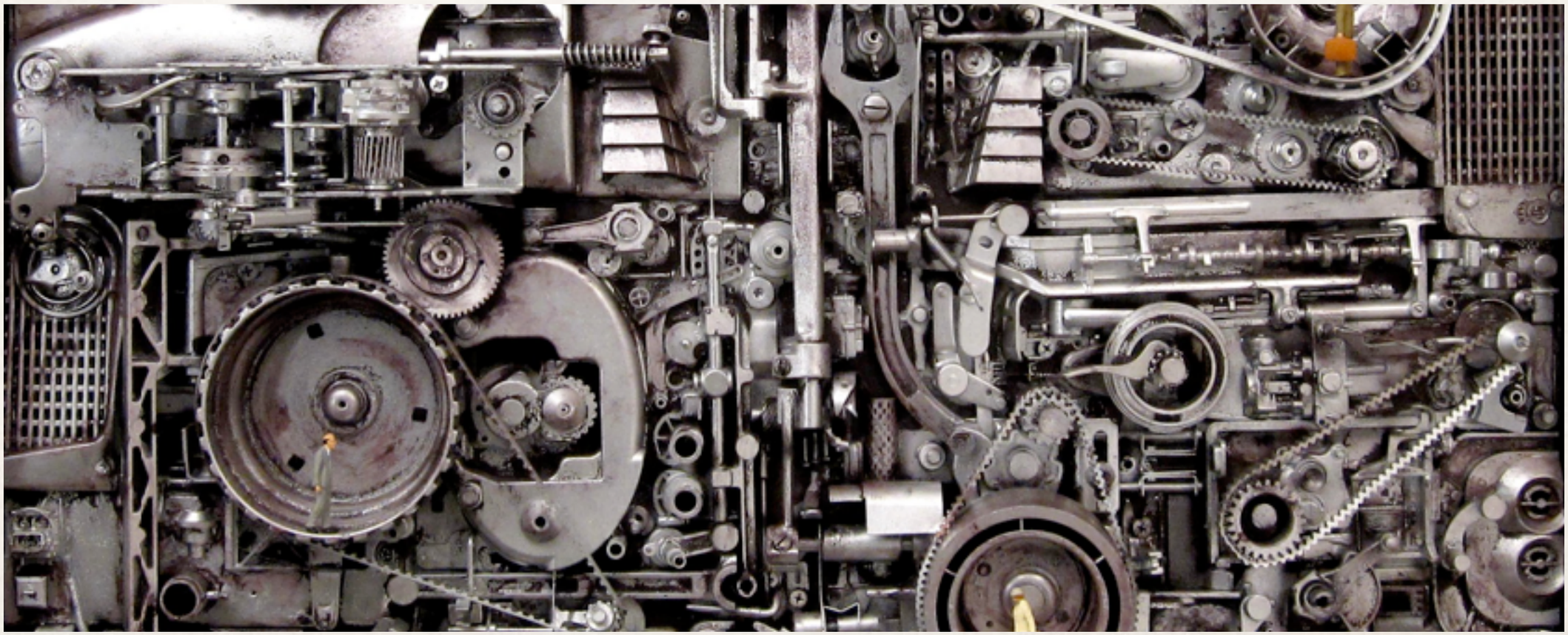
Course Structure: Exams

- **Prelims** We will have two short, in-class prelims during the semester. They are "closed-book" and "closed note" exams.
  - Wednesday, February 15
  - Wednesday, March 29
- **Final exam** We will have a 2-hour final during the University's final exam period.

Grades

- Grades are based on:
  - Problem sets: 2/9
  - Lab assignments: 1/9
  - Prelims: first 1/9; second 2/9
  - Final: 3/9
- Grades are curved. You can expect each letter grade to reflect roughly one standard-deviation. *We expect outstanding student performance in this class, and have no preconceptions about the median grade.*

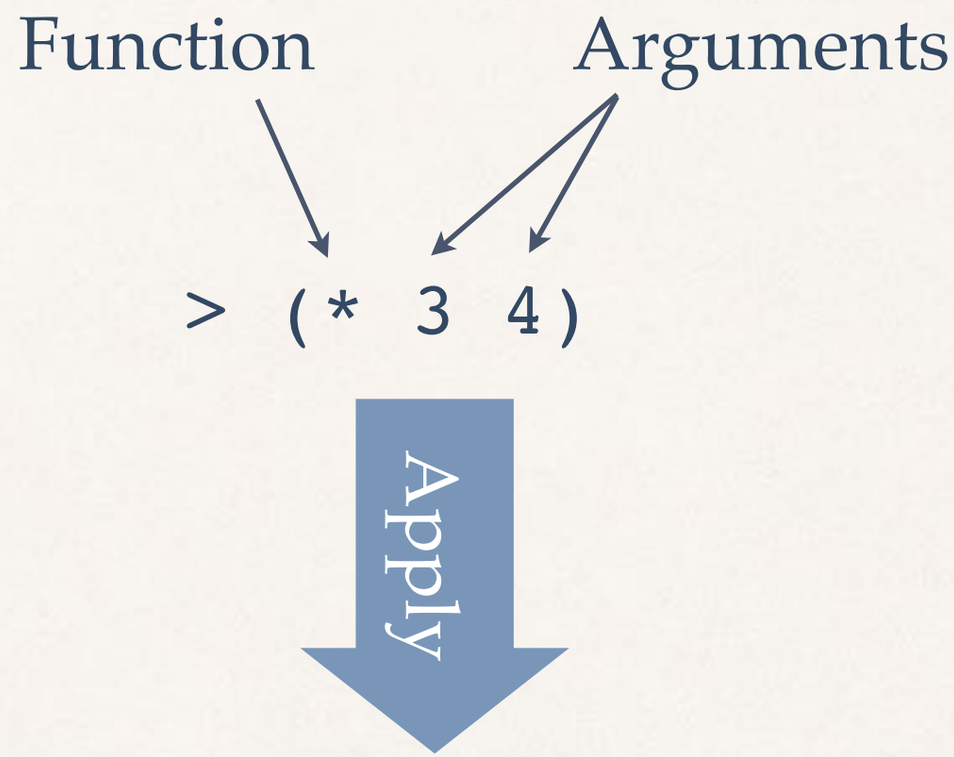
Let’s get this party started



Scheme is an *interpreted* language

- Scheme programs are built during a "dialog" with the *interpreter*.
- The interpreter's job is to
  - ***evaluate scheme expressions***, and
  - ***maintain an "environment"*** of bindings of variable to values.

The basic step



Prefix notation

- You are used to various notations for mathematical function application:
  - **prefix** notation: f(a, b). The name of the function (f) appears *before* its arguments.
  - **infix** notation: a + b. The name of the function (+) appears *between* the arguments.
  - **postfix** notation (a la HP calculators). Here the name of the function appears *after* the arguments.
- Note that prefix & postfix are naturally more suited for functions that do not take two arguments.

SCHEME uses prefix notation

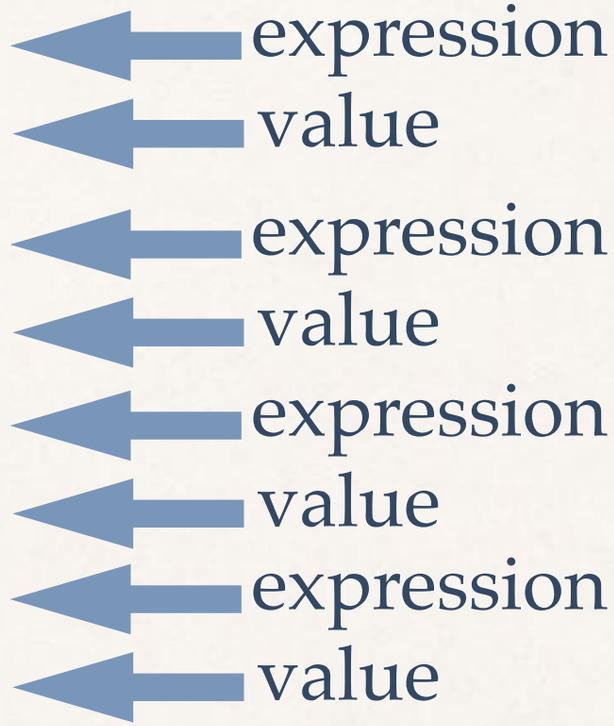
Evaluation in detail: Atomic objects

- A *number*, like 7, evaluates to...itself!
- A *function*, like +, evaluates to...itself!

So...

Examples

```
> 7
7
> 11
11
> +
#<procedure:+>
> *
#<procedure:*>
```



Evaluation in detail: Compound objects

- In SCHEME, "compound" expressions have the form

```
(<f> <a1> <a2> ... <an>)
```

To evaluate a compound expression, the interpreter

- **evaluates each subexpression and**, then,
- **applies the first,** ***which must be a function*****, to the rest,** ***which are treated as arguments to the function.***

Examples

```
> 7
7
> 11
11
> +
#<procedure:+>
> *
#<procedure:*>
> (+ 7 8)
15
> (* 18 11)
198
>
```

expression
value
expression
value
expression
value
expression
value
expression
value

This is the new part!
Argument *evaluation*
followed by
Function *application*

THE EVALUATION RULE

- If the expression is atomic (a number, a function), the result is the object itself.
- To evaluate a compound expression:
  - 0. **Evaluate** each subexpression (*recursively*, using the same rule)
  - 1. **Apply** the value of the first subexpression, which must be a function, to the values obtained from the remaining subexpressions

Recursive evaluation in action

```
> (+   (* 4 5)   (* 4 245))
```

eval

Recursive evaluation in action

```
> (+    (* 4 5)    (* 4 245))
```

eval

eval

Recursive evaluation in action

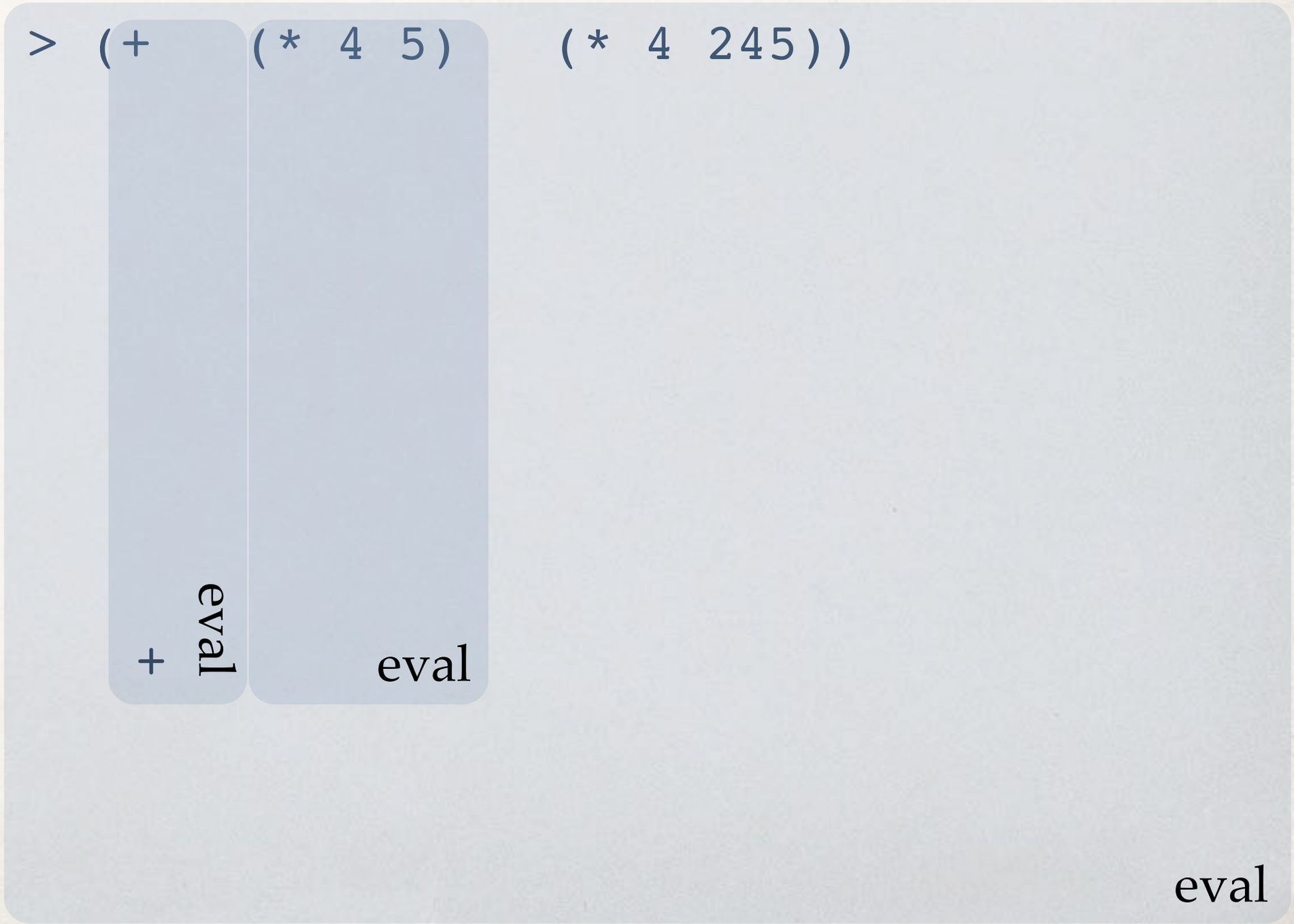
```
> (+    (* 4 5)    (* 4 245))
```

eval

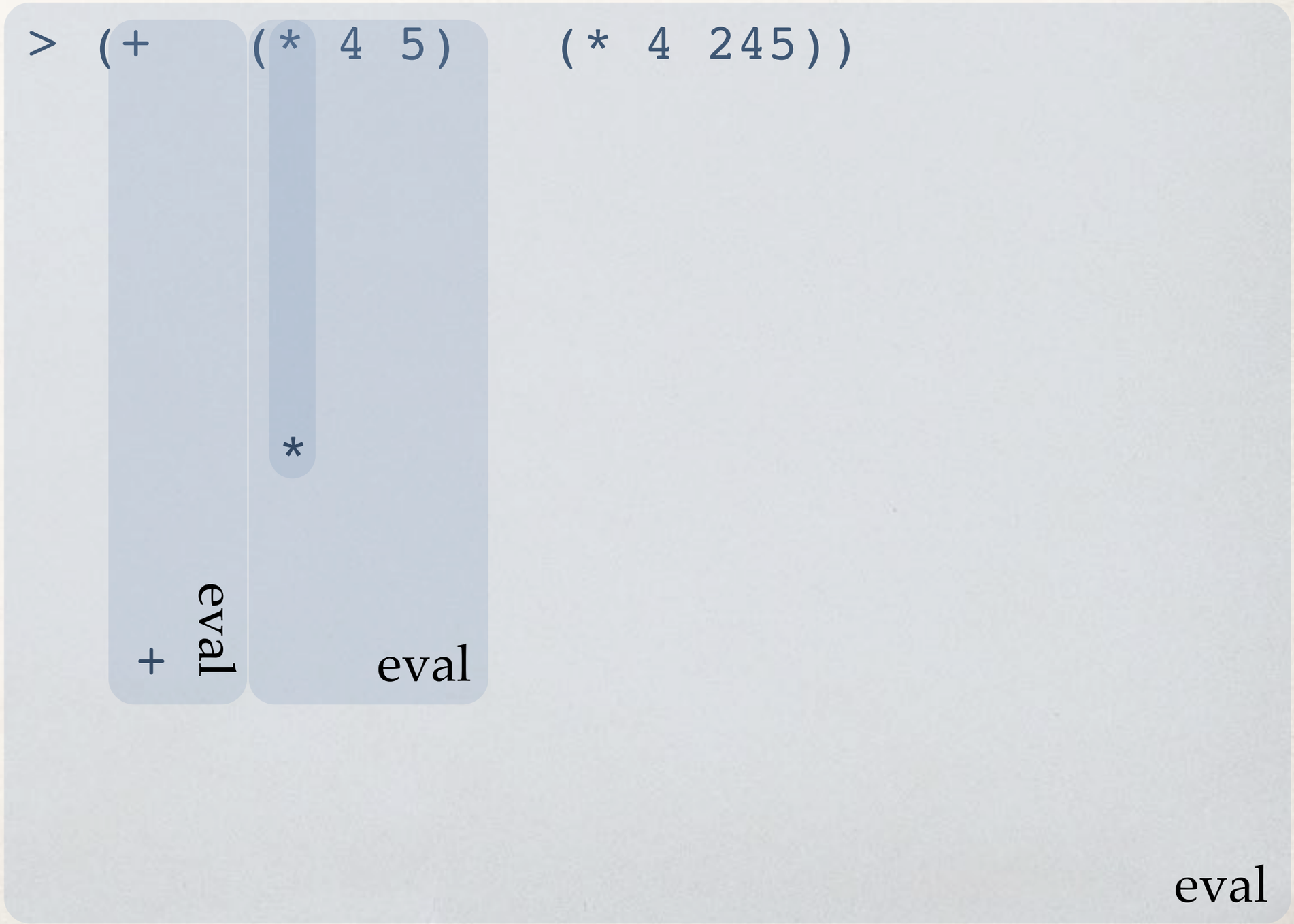
+

eval

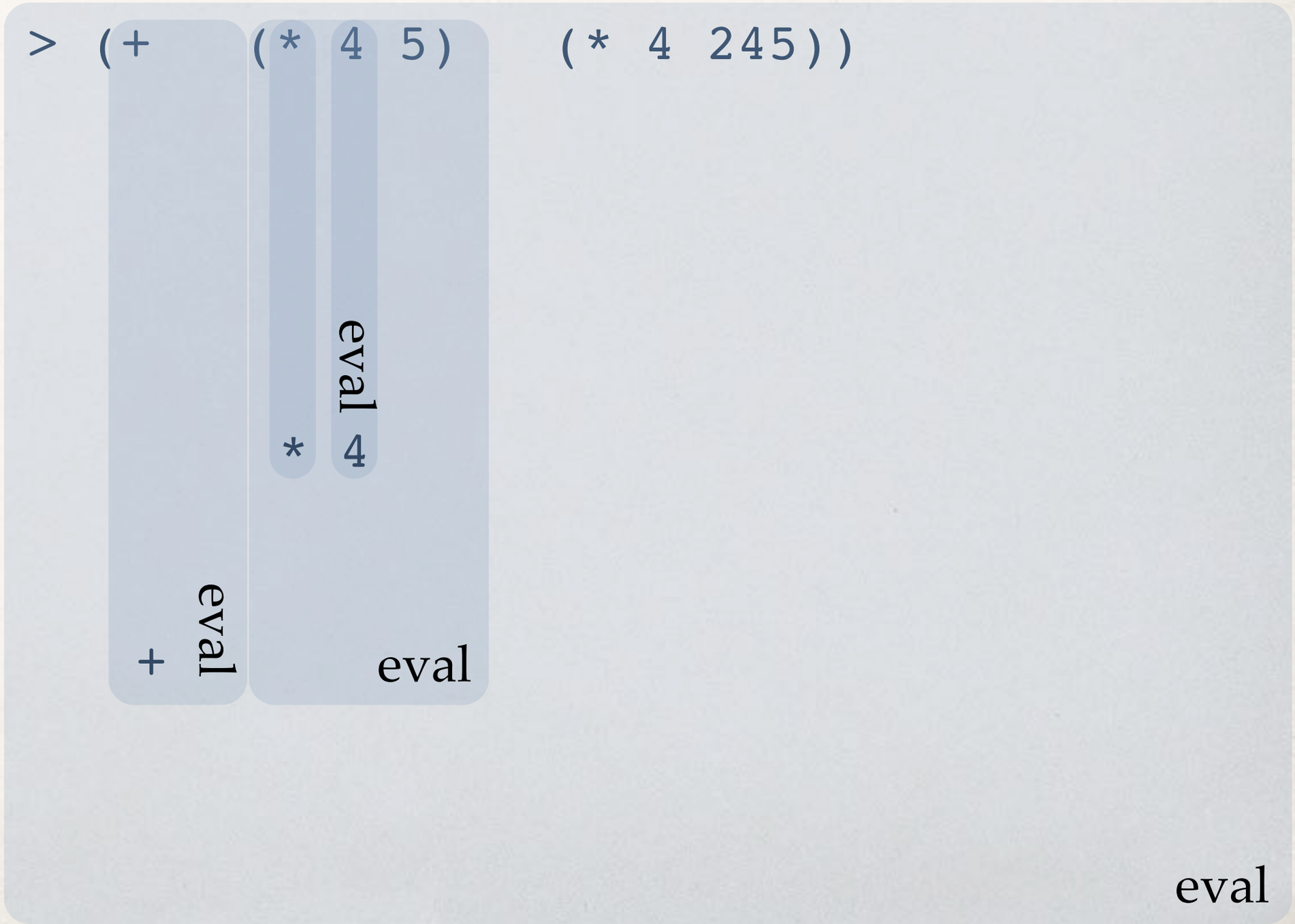
Recursive evaluation in action



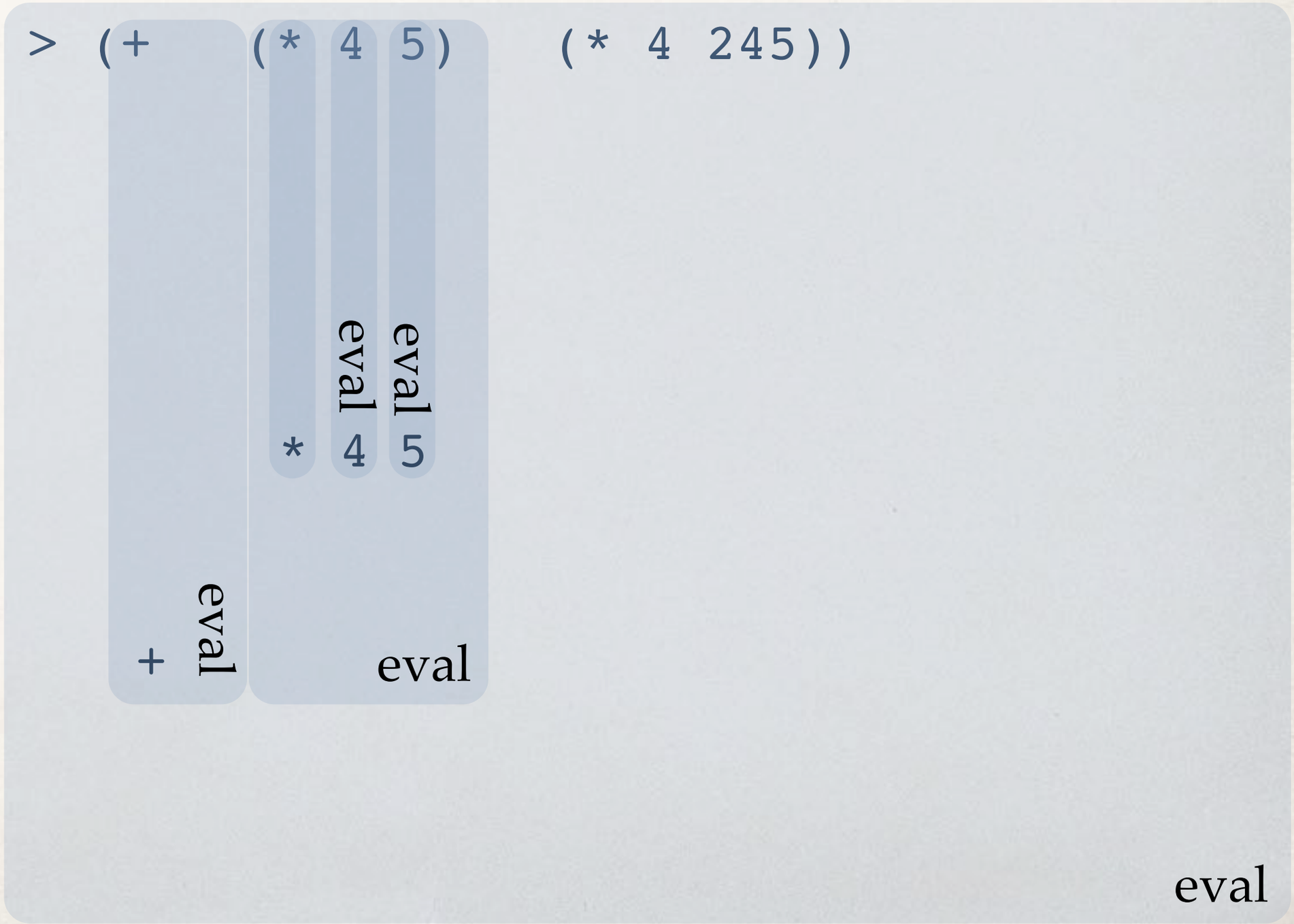
Recursive evaluation in action



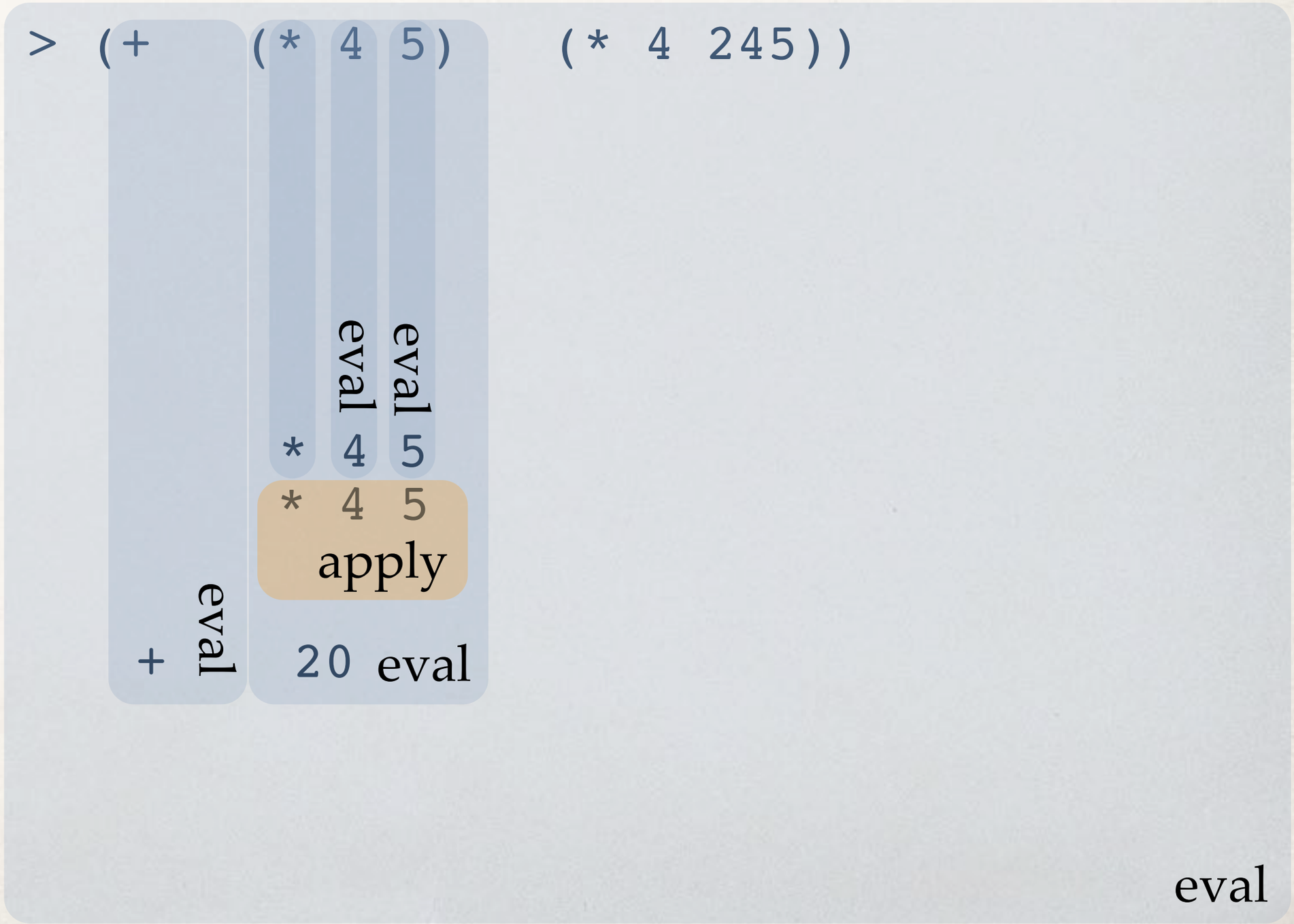
Recursive evaluation in action



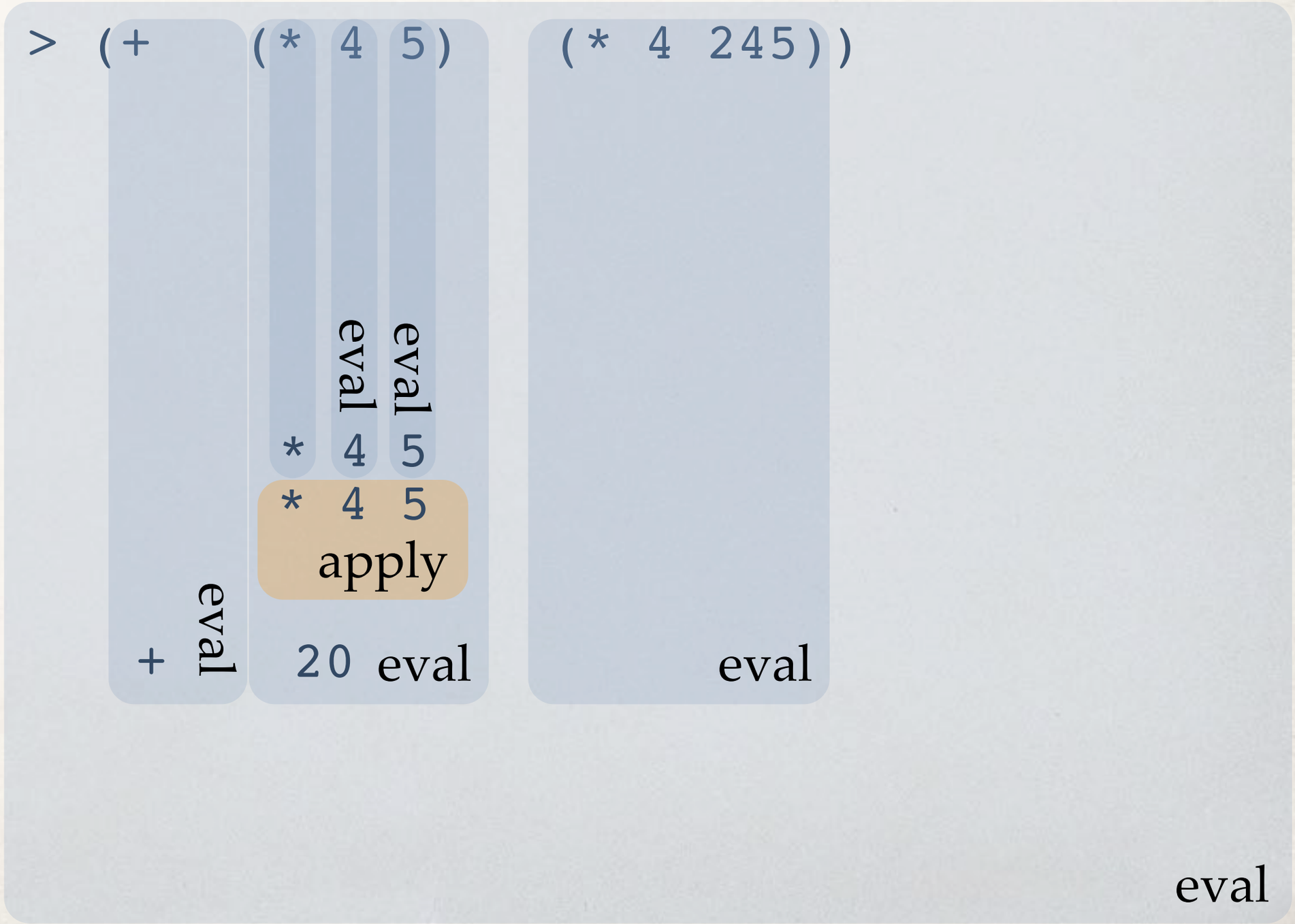
Recursive evaluation in action



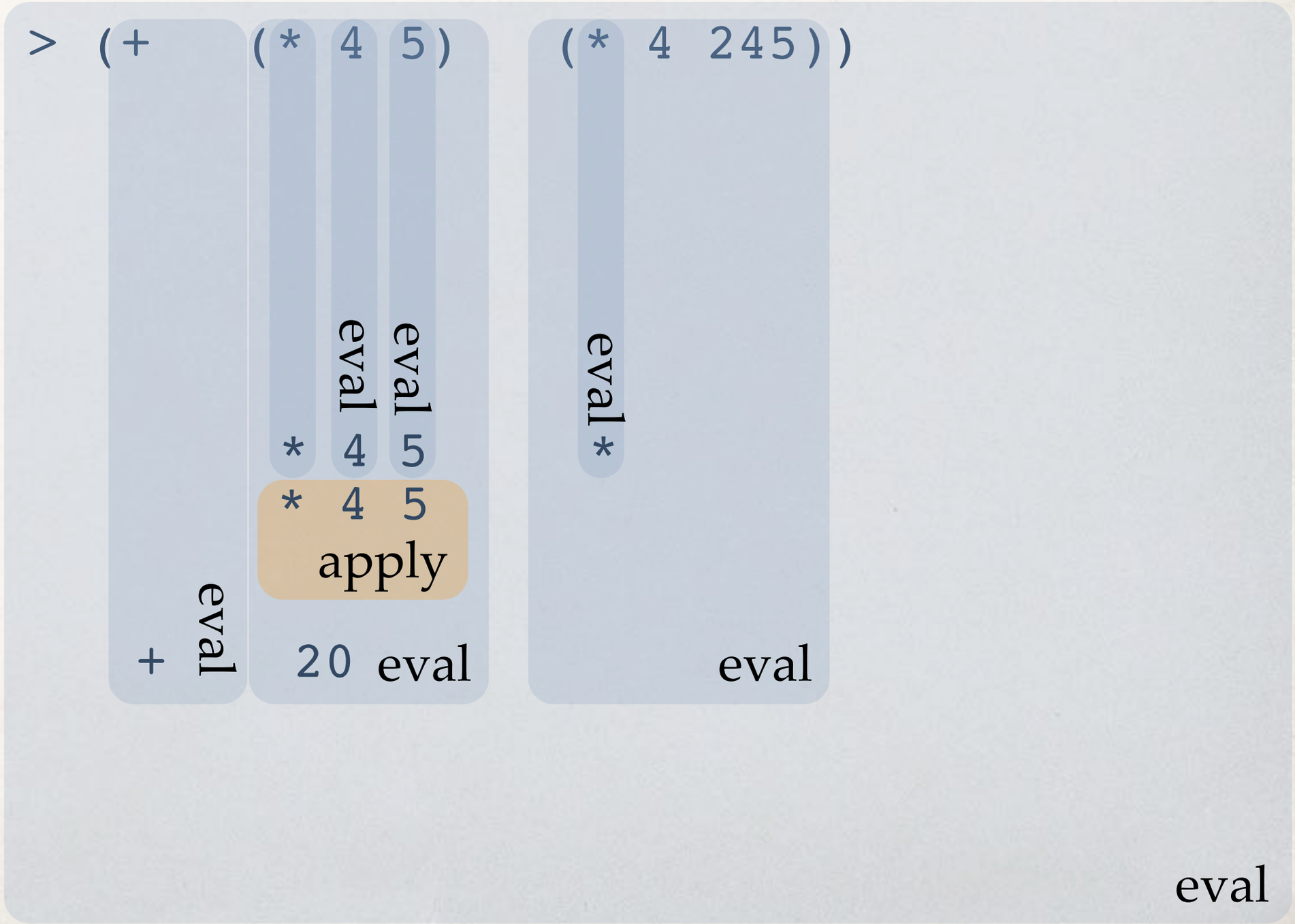
Recursive evaluation in action



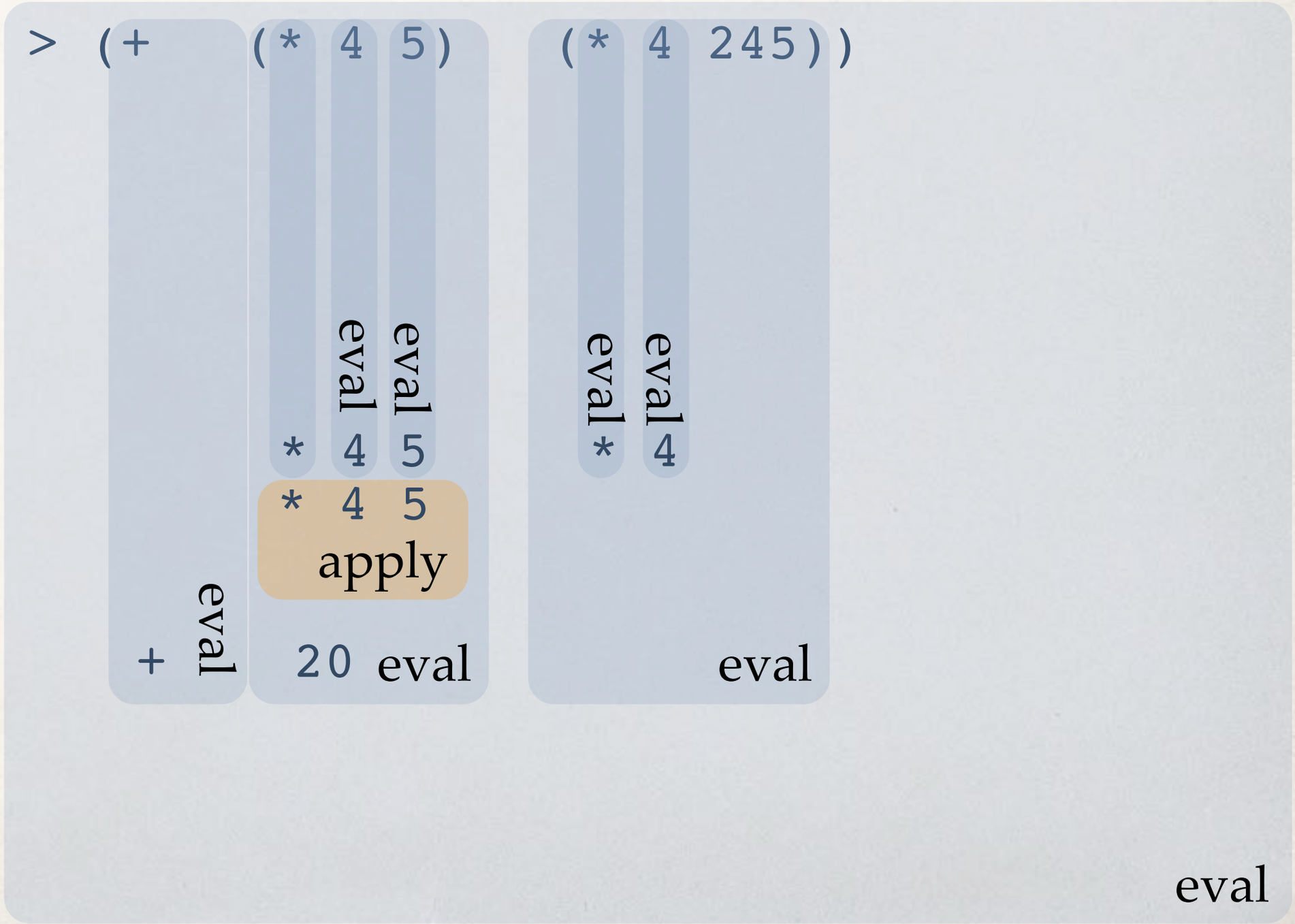
Recursive evaluation in action



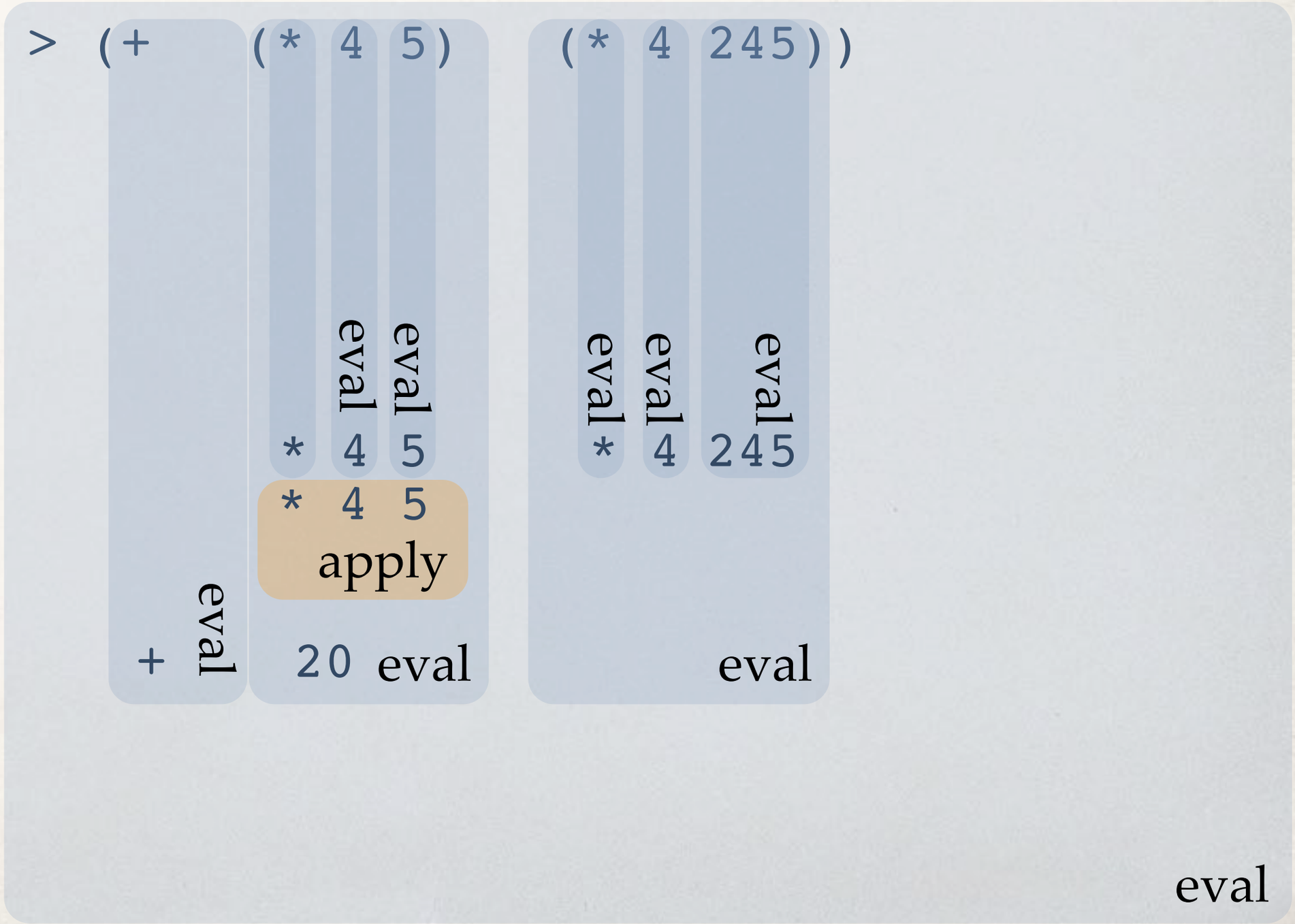
Recursive evaluation in action



Recursive evaluation in action

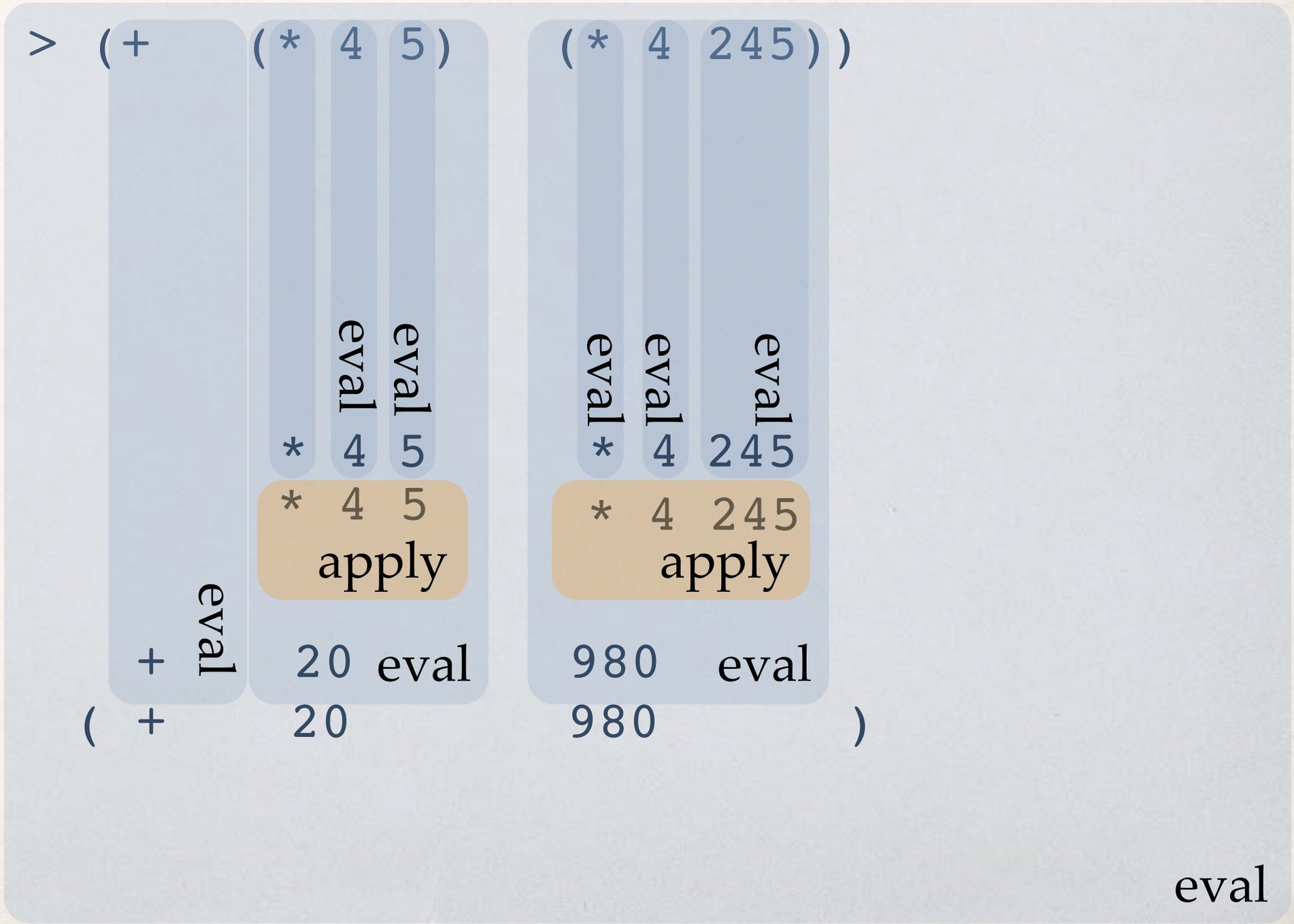


Recursive evaluation in action

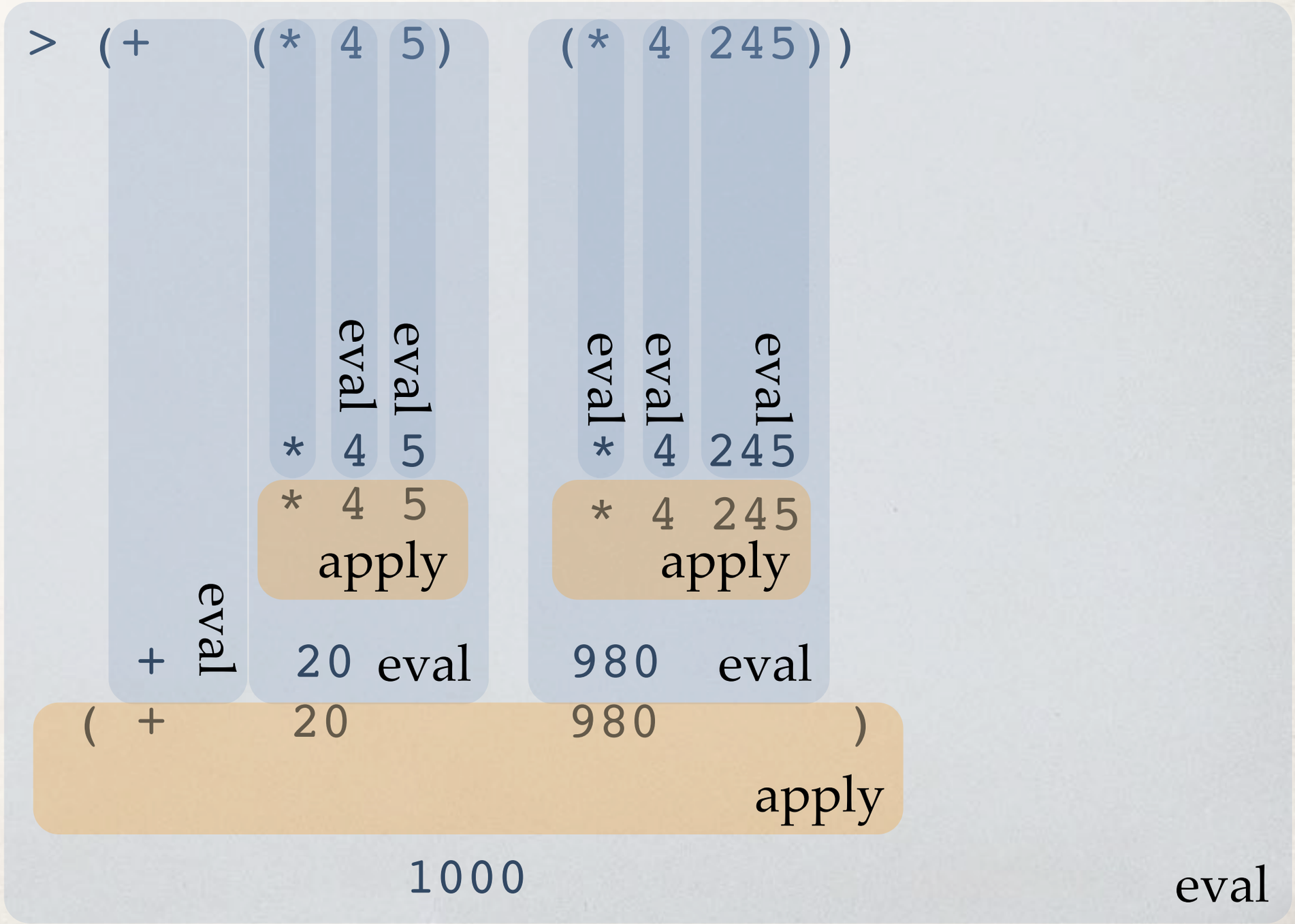


Recursive evaluation in action

Recursive evaluation in action



Recursive evaluation in action



Examples of recursive evaluation in action

```
> (+ (* 9 7) (* 45 2))
153
> (+ 5 6 7 8 9)
35
> (+ (* 9 (* 4 5)) (* 3 4))
192
>
```

Binding variables to values

- Our first example of abstraction.
- Involves a **special form**: define. (Called "special" because it breaks the regular rules of the evaluation rule.)
- The form

```
(define <variable> <value>)
```

binds the variable `<variable>` to the value obtained by evaluating `<expression>`.

So…

Binding: Examples

```
> (define a 6)
> a
6
> (+ a 4)
10
> (define b (+ 3 4))
> b
7
> (* a b)
42
> (define a 7)
> a
7
> (* a b)
49
```

Environments

- An *environment* is a family of bindings of variables to values.

```
> (define a 6)
> a
6
> (define b (+ 3 4))
> b
7
> (define a 5)
> a
5
> (* a b)
35
```

$a \longmapsto 6$

The variable a is *bound* to 6

$a \longmapsto 6$
$b \longmapsto 7$

The variable b is *bound* to 7

$a \longmapsto 5$
$b \longmapsto 7$

The variable a is *bound* to 5;
this changes the binding!

THE EVALUATION RULE (again, with environments...)

- If the expression is atomic (a number, a function), the result is the object itself.

- If the expression is a variable, the result is the value this variable is assigned in the current environment.

- To evaluate a compound expression:

  - 0. **Evaluate** each subexpression (recursively, using the same rule)

  - 1. **Apply** the value of the first subexpression, which must be a function, to the values obtained from the remaining subexpressions

Defining new **functions**

- The **define** special form can also be used to define new *functions*. The syntax is the following:

```
(define (<function-name> <argument-var>)
        <body>)
```

- Thus,

```
> (define (inc x) (+ x 1))
> (inc 1)
2
> (inc 2)
3
> (inc (inc 5))
7
```

Defines the "increment" function:
`inc`(x) = x+1

More examples of function definition

```
(define (inc x) (+ x 1))

(define (dec x) (- x 1))

(define (square x) (* x x))


(define (sum-of-squares x y)
        (+ (square x) (square y)))
```

Increment. `inc`(x) = x+1.

Decrement. `dec`(x) = x-1.

Square. `square`(x) = x^2.

This last function, `sum-of-squares`, is a function of two variables:

`sum-of-squares`(x,y) = x^2+y^2.

Function definition: *terminology*

- Consider `(define (f x) <body>)`. Here
  - `f` is the function *name*,
  - `x` is the *formal parameter*, and
  - `<body>` is the *body* of the function, which usually contains references to the variable `x` (the parameter).
- When the function is called (e.g., `(f 8)`), the value it is called upon (8, in this case) is the *actual parameter*.
- The word *argument* is also used, for both the formal and actual parameter.

Substitution semantics: A simple model of function application

- Consider `(define (f x) <body>)`.
- In the future, if the interpreter is called upon to evaluate the expression `(f <expr>)` it will:
  - 1. (As usual) evaluate the expression `<expr>`, resulting in a value v
  - 2. substitute each occurrence of `x` in `<body>` with the value v.
  - 3. return the result of evaluating `<body>` (after substitution).

An example

```
(define (inc x) (+ x 1))
```

The definition

```
(inc (+ 4 5))
```

The invocation

How is the invocation evaluated?

According to substitution semantics:

- The actual argument `(+ 4 5)`, is evaluated which yields the value 9.
- 9 is substituted for `x` in the body of the function

```
(+ x 1)
```

which yields `(+ 9 1)`. This is evaluated to 10, which is returned.

The reality is a little different: **Environment semantics** (We will discuss in detail later)

- Consider `(define (f x) <body>)`.

- In the future, if the interpreter is called upon to evaluate the expression `(f <expr>)` it will:

  - (As usual) evaluate the expression `<expr>`, resulting in a value v

  - create a *new environment*, identical to the current one but in which `x` has been bound to v.

  - evaluate the expression `<body>` in this new environment; the resulting value of `<body>` is the value this function returns.

Example again

```
(define (inc x) (+ x 1))
```

The definition

```
(define a 4)
(define b 5)
(inc (+ a b))
```

The invocation

$a \longmapsto 4$

The variable a is *bound* to 4

Example again

```
(define (inc x) (+ x 1))

(define a 4)
(define b 5)
(inc (+ a b))
```

The definition

The invocation

$a \longmapsto 4$

$b \longmapsto 5$

The variable a is *bound* to 4
The variable b is *bound* to 5

Example again

```
(define (inc x) (+ x 1))
```

The definition

```
(define a 4)
(define b 5)
```

The invocation

```
(inc (+ a b))
```

Argument is evaluated

$a \longmapsto 4$

$b \longmapsto 5$

The variable a is *bound* to 4
The variable b is *bound* to 5

9

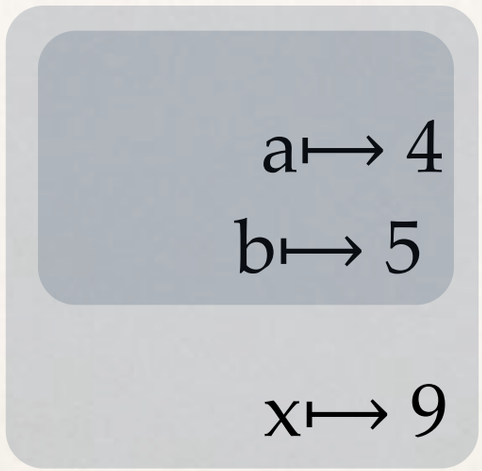
Example again

```
(define (inc x) (+ x 1))

(define a 4)
(define b 5)
(inc (+ a b))
```

The definition

The invocation



The variable a is *bound* to 4
The variable b is *bound* to 5

Formal *x* is *bound* to 9

Example again

```
(define (inc x) (+ x 1))

(define a 4)
(define b 5)
(inc (+ a b))
```

The definition

The invocation

Body is evaluated in new environment!

$a \longmapsto 4$

$b \longmapsto 5$

$x \longmapsto 9$

The variable a is *bound* to 4
The variable b is *bound* to 5

Formal *x* is *bound* to 9

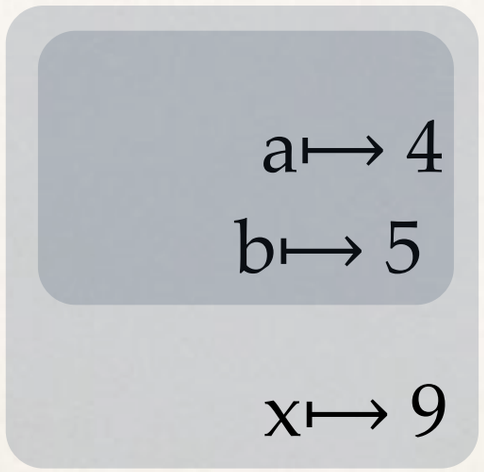
Example again

```
(define (inc x) (+ x 1))

(define a 4)
(define b 5)
(inc (+ a b))
```

The definition

The invocation



The variable a is *bound* to 4
The variable b is *bound* to 5

Formal *x* is *bound* to 9

Value to be returned....

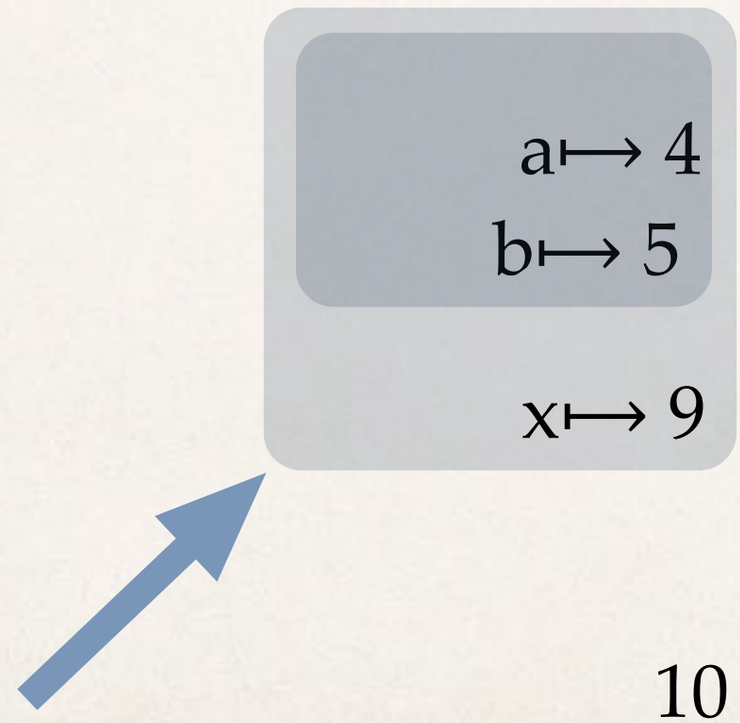
Example again

```
(define (inc x) (+ x 1))

(define a 4)
(define b 5)
(inc (+ a b))
```

The definition

The invocation



The variable a is *bound* to 4
The variable b is *bound* to 5

Formal *x* is *bound* to 9

10

Value to be returned....

the “call environment” is discarded!

Example again

```
(define (inc x) (+ x 1))

(define a 4)
(define b 5)
(inc (+ a b))
```

The definition

The invocation

$a \longmapsto 4$

$b \longmapsto 5$

The variable a is *bound* to 4
The variable b is *bound* to 5

Formal *x* is *bound* to 9

Value returned to interpreter

Boolean values in SCHEME

- Along with numbers, which we've already explored, SCHEME can maintain Boolean values (true/false). These are denoted `#t` and `#f`.

- As with numbers, they evaluate to themselves.

- Scheme has a full set of logical functions:

  - `(and x y)`, true exactly when both `x` and `y` are true,

  - `(or x y)`, true when either `x` or `y` is true (or both),

  - `(not x)`, returns of negation of its argument.

Examples

```
> (define a #t)
> (and a #f)
#f
> (or a #f)
#t
> (and a a)
#t
> (not a)
#f
> (or #f (and #t #t))
#t
```

Numeric comparison

- Scheme provides comparison functions for numbers. These return Boolean values.

- `(= x y)`, returns `#t` (true) when `x` is equal to `y`, otherwise `#f`.

- `(> x y)`, `(< x y)`, returns true when `x` is greater than (less than) `y`,

- `(>= x y)`, `(<= x y)`, greater than (less than) or equal.

Conditionals via `if`

- Simple conditionals are implemented by the `if` special form. The syntax is

```
(if <pred>
    <then-clause>
    <else-clause>)
```

- The expression `<pred>` is evaluated.

- If it evaluates to `#t`, `<then-clause>` is evaluated, and its value is returned. Otherwise, `<else-clause>` is evaluated, and its value is returned.

- Note that *only one of the two* expressions `<then-clause>` and `<else-clause>` is evaluated. (This requires `if` to be a special form.)

Examples, `if` you please...

```
> (if #t 3 4)
3
> (if #f 3 4)
4
> (if (> 3 4)
      "Three is bigger than four"
      "Three is not bigger than four")
"Three is not bigger than four"
> (define (geq3 x)
    (if (>= x 3) "At least three" "Less than three"))
> (geq3 4)
"At least three"
> (geq3 3)
"At least three"
> (geq3 2)
"Less than three"
>
```

Conditionals via the `cond` special form

- The SCHEME *conditional* special form:

```
(cond (<guard1> <expr1>)
      (<guard2> <expr2>)
      ...
      (<guardn> <exprn>))
```

- Evaluates the expressions `<guard1>`, `<guard2>`, ... in this order until it finds one, say `<guardk>`, that evaluates to `#t`. Then returns the value obtained by evaluating `<exprk>`.

- NOTE: *Only one* of the `<exprk>` is evaluated. This is important, as we shall see in future lectures.

- We do not define the result if none of the guards evaluate to `#t`.

The `else` guard in `cond`

- You can use `else` as a guard that always evaluates to true. (You might call this "syntactic sugar" for `#t`.)

```
> (cond (#f 1)
        (else 2))
2
```

- Equivalently, you could use `#t`.

```
> (cond (#f 1)
        (#t 2))
2
```

Conditional examples

```
> (cond (#f 3)
        (#t 4)
        (#t 5)
        (#f 6))
4
> (cond ((> 4 5) 1)
        ((> 4 3) 2))
2
> (if (> 4 3) 7 8)
7
> (if (<= 4 3) 7 8)
8
>
```

Some simple examples

```
> (define (absolute x)
          (if (>= x 0)
              x
              (* -1 x)))
> (absolute 7)
7
> (absolute -7)
7
>
```

```
(define (hailstone n)
        (if (= (modulo n 2) 0)
            (/ n 2)
            (+ (* 3 n) 1)))
```

```
> (hailstone 6)
3
> (hailstone 3)
10
> (hailstone 10)
5
> (hailstone 5)
16
> (hailstone 16)
8
> (hailstone 8)
4
> (hailstone 4)
2
> (hailstone 2)
1
> (hailstone 1)
4
> (hailstone 4)
2
```

Reading

- Read "The Elements of Programming," Section 1.1. of SICP.
- You might enjoy reading about the hailstone problem.

http://en.wikipedia.org/wiki/Collatz_conjecture