CS 152

Programming Paradigms

Functional Programming

https://xkcd.com/1270/

Today

- Advantages & Popularity of Functional Programming
- Characteristics of Functional Programming
- Scheme: A Dialect of Lisp

Course Learning Outcomes

2. Have a basic knowledge of the procedural, object-oriented, functional, and logic programming paradigms.
11. Produce programs in a functional programming language in excess of 200 LOC.

Background

- Until recently, most functional languages suffered from inefficient execution
- Most were originally interpreted instead of compiled

Advantages

- Today, functional languages are very attractive for general programming
- They lend themselves very well to parallel execution
- They may be more efficient than imperative languages on multicore hardware architectures
- They have mature application libraries making them suitable for implementing complex systems

Advantages

- Functional programming languages generally have simpler semantics and a simpler model of computation
- Useful for rapid prototyping, artificial intelligence, mathematical proof systems, and logic applications

Popularity?

- Despite these advantages, functional languages have not become mainstream languages. Why?
- Programmers learn imperative or object-oriented languages first.
- OO languages provide an intuitive way for structuring code that mirrors the everyday experience of real objects.
- Functional programming provide a more abstract and mathematical mechanism for structuring code => higher barrier to entry.

Functions Everywhere

- Functional methods such as recursion, functional abstraction, and higher-order functions have become part of many programming languages.

Main Characteristics

- Provide a uniform view of programs as functions
- Treat functions as data
- Prevent side effects

Programs as Functions

- A program is a description of specific computation
- If we ignore the "how" and focus on the result, or the "what" of the computation:

- A program is essentially equivalent to a mathematical function

Mathematical Function

A function is a rule that associates with *x* from the set *X* of values **a unique** *y* from the set *Y* of values

$$y = f(x)$$

$$f\colon X \rightarrow Y$$

x: **independent** variable

y: **dependent** variable

Functions Everywhere

- Programs, procedures, and functions can all be represented by the mathematical concept of a function
- At the program level, *x* represents the input, and *y* represents the output
- At the procedure or function level, *x* represents the parameters, and *y* represents the return values

Functions

- Function Definition
- Function Application

Function Definition

Function definition: describes how a return value is to be computed using formal parameters.

In Python:

```
def increment(x):
    return 1 + x
```

In lambda calculus: $(\lambda\, x.\, +\, 1\; x)$

In Scheme (anonymous function definition): `(lambda (x) (+ 1 x))`

Function Application

Function application: a call to a defined function using actual values.

In Python: increment(5)

In lambda calculus: $(\lambda\, x. + 1\; x)$ 5

Represents the application of "the function that adds 1 to x" to the constant 5

In Scheme:

((lambda (x) (+ 1 x)) 5)

Variables

- In imperative programming languages, variables refer to memory locations that store values
- In Math, variables always stand for actual values

A major difference between imperative programming and functional programming is the concept of a variable

Assignments

- Assignment statements allow memory locations to be reset with new values
- In Math, there are no concepts of memory location and assignment

Value Semantics

- Functional programming takes a mathematical approach to the concept of a variable
- Variables are bound to values, not memory locations
- Value semantics: semantics in which names are associated only with values, not memory locations
- A variable’s value cannot change, which eliminates assignment as an available operation

Variables & Assignments

- Most functional programming languages retain some notion of assignment
- A pure functional program takes a strictly mathematical approach to variables => no assignment

Loops

- Lack of assignment makes loops impossible
- A loop requires a control variable whose value changes as the loop executes
- Recursion is used instead of loops

State

- There is no notion of the internal state of a function
- The return value depends only on the values of its arguments (and possibly nonlocal variables)
- A function’ s value cannot depend on the order of evaluation of its arguments
- An advantage for concurrent applications

Referential Transparency

A function is referentially transparent if:

- The function’s return value depends only on the values of its arguments
- The function's application (call) can be replaced by the return value without changing the program's behavior (no side effect)

iClicker: Referential Transparency

- The function’s return value depends only on the values of its arguments
- The function's application (call) can be replaced by the return value without changing the program's behavior (no side effect)

```
def double(number):
    result = number * 2
    return result
```

Is this function referentially transparent?

A. Yes

B. No

iClicker: Referential Transparency

- The function’s return value depends only on the values of its arguments
- The function's application (call) can be replaced by the return value without changing the program's behavior (no side effect)

```
count = 0
def mystery1(number):
    global count
    result = number + count
    count += 1
    return result
```

Is this function referentially transparent?

A. Yes
B. No

Referential Transparency

```
count = 0
def mystery1(number):
    global count
    result = number + count
    count += 1
    return result
print(mystery1(3))
print(mystery1(3))
print(mystery1(3))
```

3
4
5

Referential Transparency

```
count = 0
def mystery2(number):
    global count
    result = number * 2
    count += 1
    return result
```

Does this function have side effects?

A. Yes
B. No

Functional Programming

```
count = 0
def mystery2(number):
    global count
    result = number * 2
    count += 1
    return result
```

Is this function referentially transparent?

A. Yes
B. No

Functional Programming

```
count = 0
def mystery2(number):
    global count
    result = number * 2
    count += 1
    return result
print(mystery2(5))
print(count)
```

```
10
1
```

Can we replace a function call such as mystery2(5) by the return value (10) without changing the program's behavior?

Functional Programming

```
count = 0
def mystery2(number):
    global count
    result = number * 2
    count += 1
    return result
print(10)
print(count)
```

10
0

Can we replace a function call such as mystery2(5) by the return value (10) without changing the program's behavior?

Pure Functions

- Pure functions, with no side effects are referentially transparent.

Referential Transparency

A referentially transparent function with no parameters?

A. Does not exist

B. Can return any value

C. Behaves like a constant

Referential Transparency & Concurrency

A function’ s return value cannot depend on the order of evaluation of its arguments:

multiply(add(3, 4), subtract(6, 5))

add(3, 4) and subtract(6, 5) are called

Their return values are used in the call to *multiply:*

multiply(7, 1)

Choice: Which function is called first: add(3, 4) or subtract (6, 5)

Question: Does it matter?

Referential Transparency & Concurrency

```
seed = 5
def mystery(number):
    global seed
    seed += 1
    return seed + number

def add_them(a, b):
    return a + b

print(add_them(seed, mystery(1)))
```

What is printed if the function arguments are evaluated from left to right?

Referential Transparency & Concurrency

```
seed = 5
def mystery(number):
    global seed
    seed += 1
    return seed + number

def add_them(a, b):
    return a + b

def main():
    print(add_them(seed, mystery(1)))
```

What would be printed IF the function arguments were evaluated from right to left?

Referential Transparency & Concurrency

- If there are no side effects, the order of evaluation of subexpressions will make no difference
- If there are side effects, there may be differences

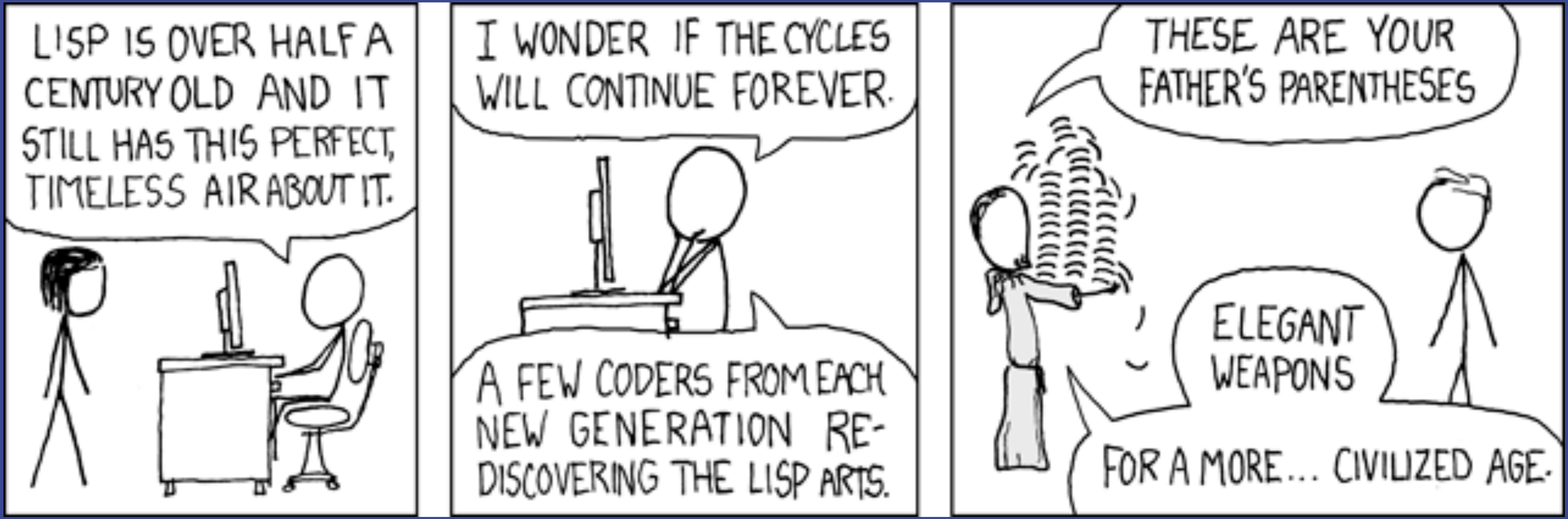
First Class Functions

- Functions are general language objects, viewed as values themselves (first-class data values)
- Functions can be passed as arguments to other functions.
- Functions can be created dynamically and returned by other functions.

Recap: Functional Programming

- All procedures are functions that distinguish incoming values (parameters/independent variable) from outgoing values (results/dependent variable)
- In pure functional programming, there are no assignments
- In pure functional programming, there are no loops
- Value of a function depends only on its arguments, not on order of evaluation or execution path
- Functions are first-class data values

Scheme: A Dialect of Lisp



https://xkcd.com/297/

Scheme, A Dialect of Lisp

'the only computer language that is beautiful'
-Neal Stephenson

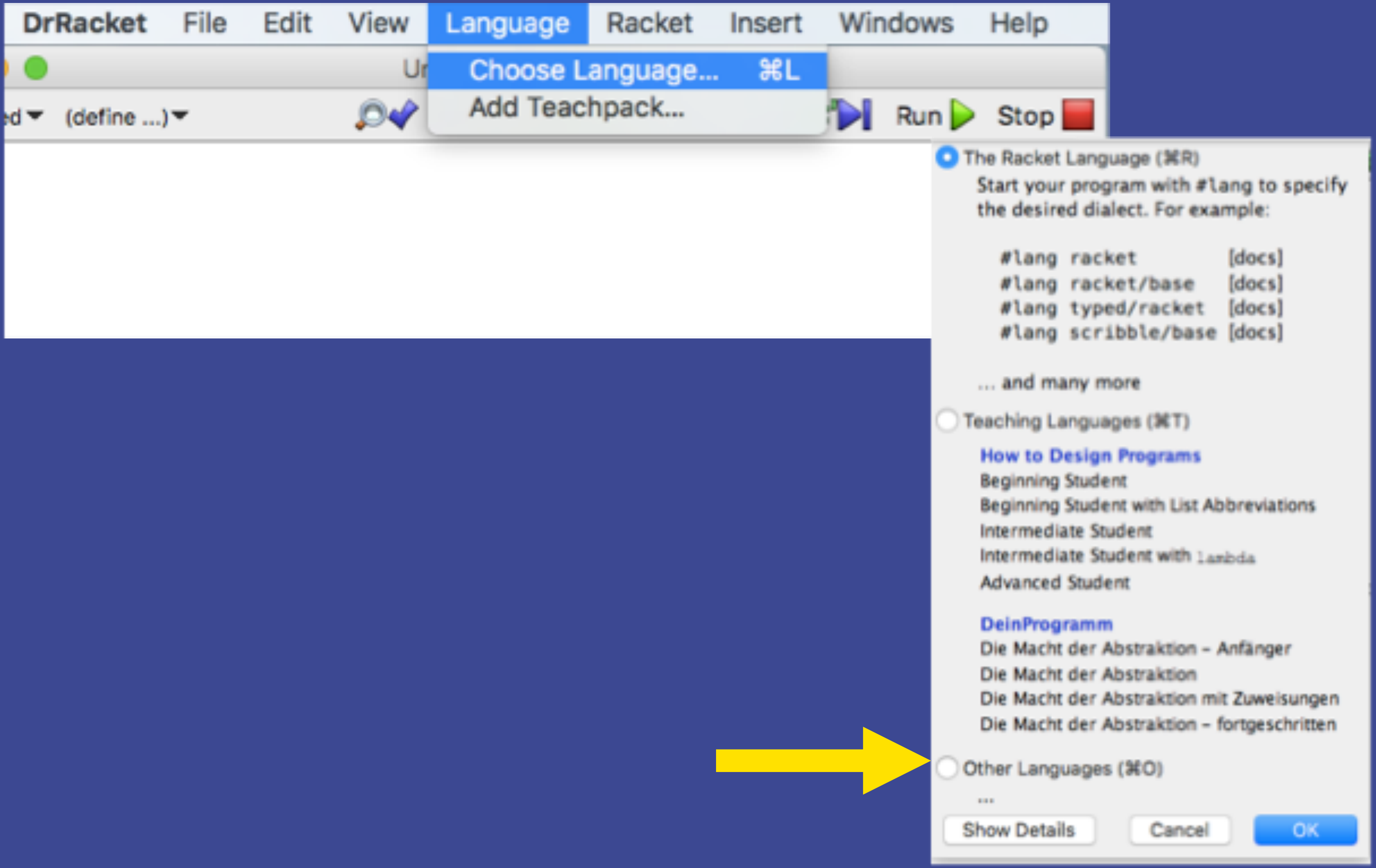
Scheme: A Dialect of Lisp

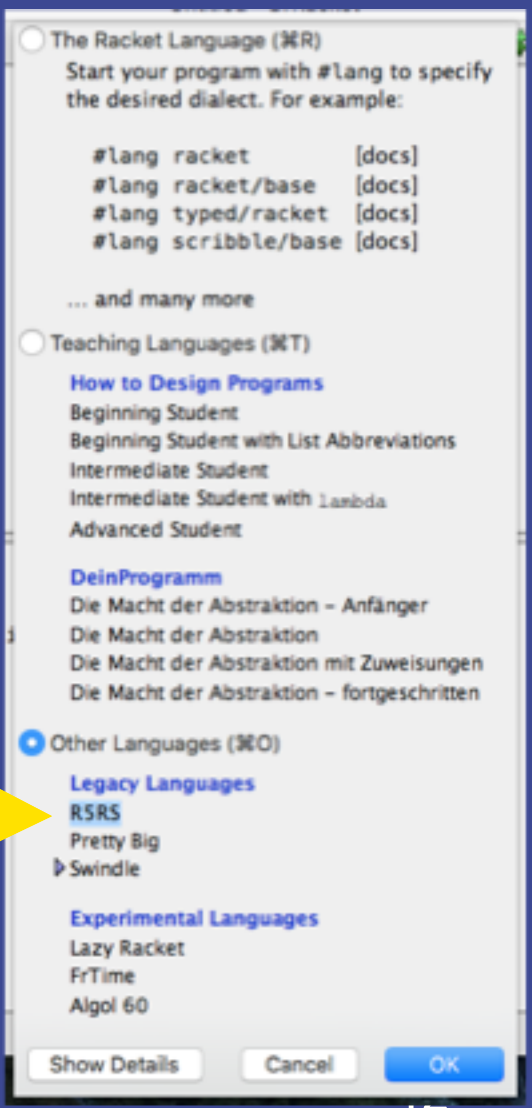
- Lisp (LISt Processing): first language that contained many of the features of modern functional languages
  - Based on lambda calculus
- Features included:
  - Uniform representation of programs and data using a single general structure: the list
  - Definition of the language using an interpreter written in the same language (metacircular interpreter)
  - Automatic memory management by the runtime system

Scheme: A Dialect of Lisp

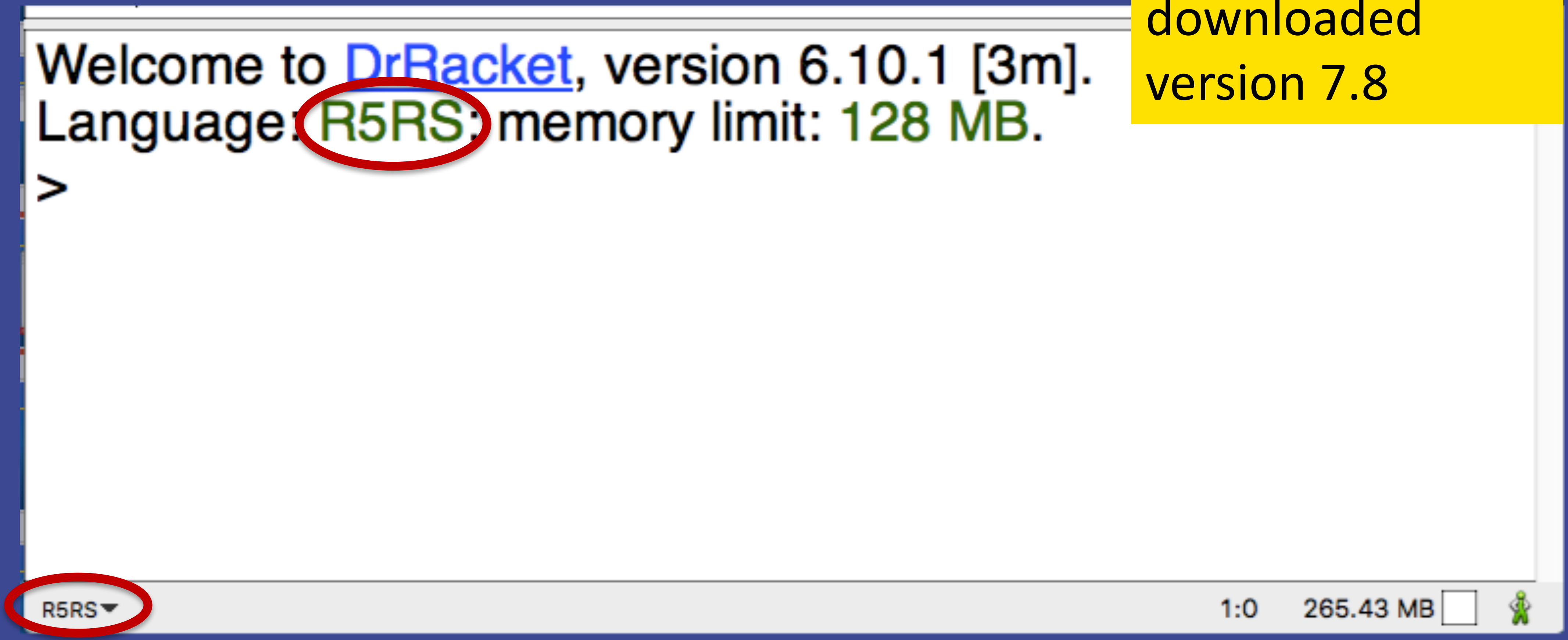
- No single standard evolved for Lisp, and there are many variations
- Two dialects that use static scoping and a more uniform treatment of functions have become standard:
  - Common Lisp
  - Scheme
  - All major Scheme dialects implement the R5RS specification

Scheme R5RS in DrRacket



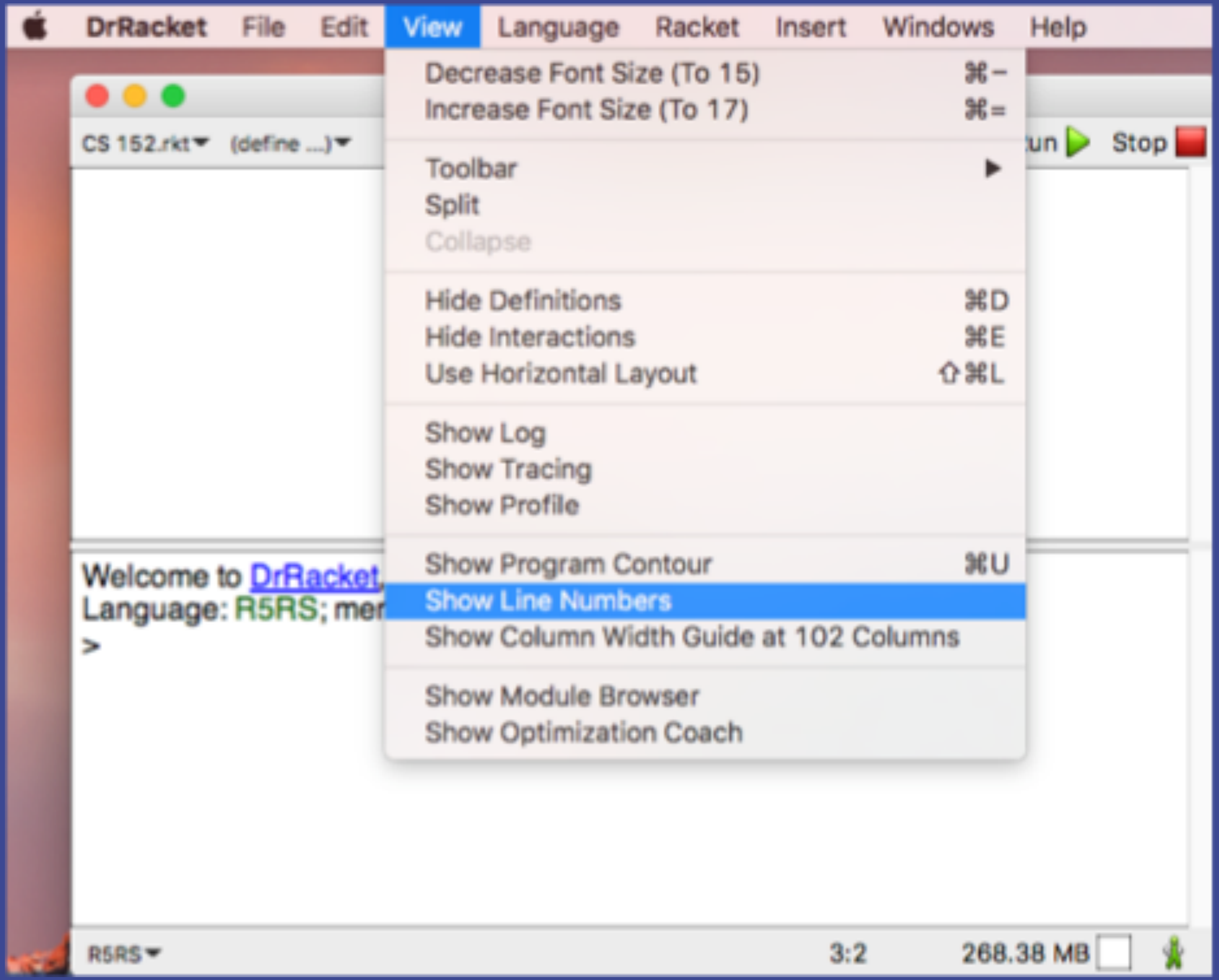


Scheme R5RS in DrRacket

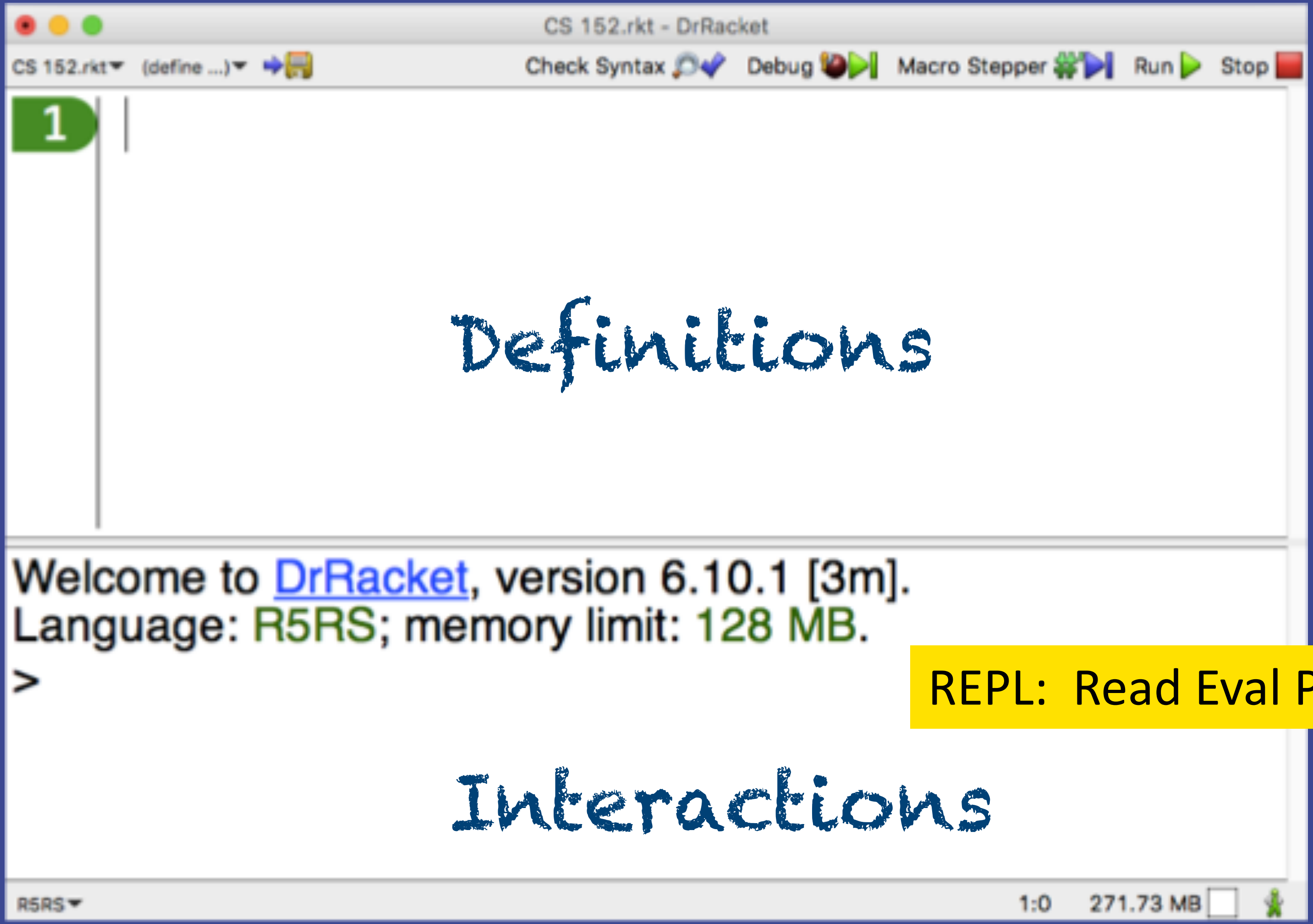


You have downloaded version 7.8

Scheme R5RS in DrRacket

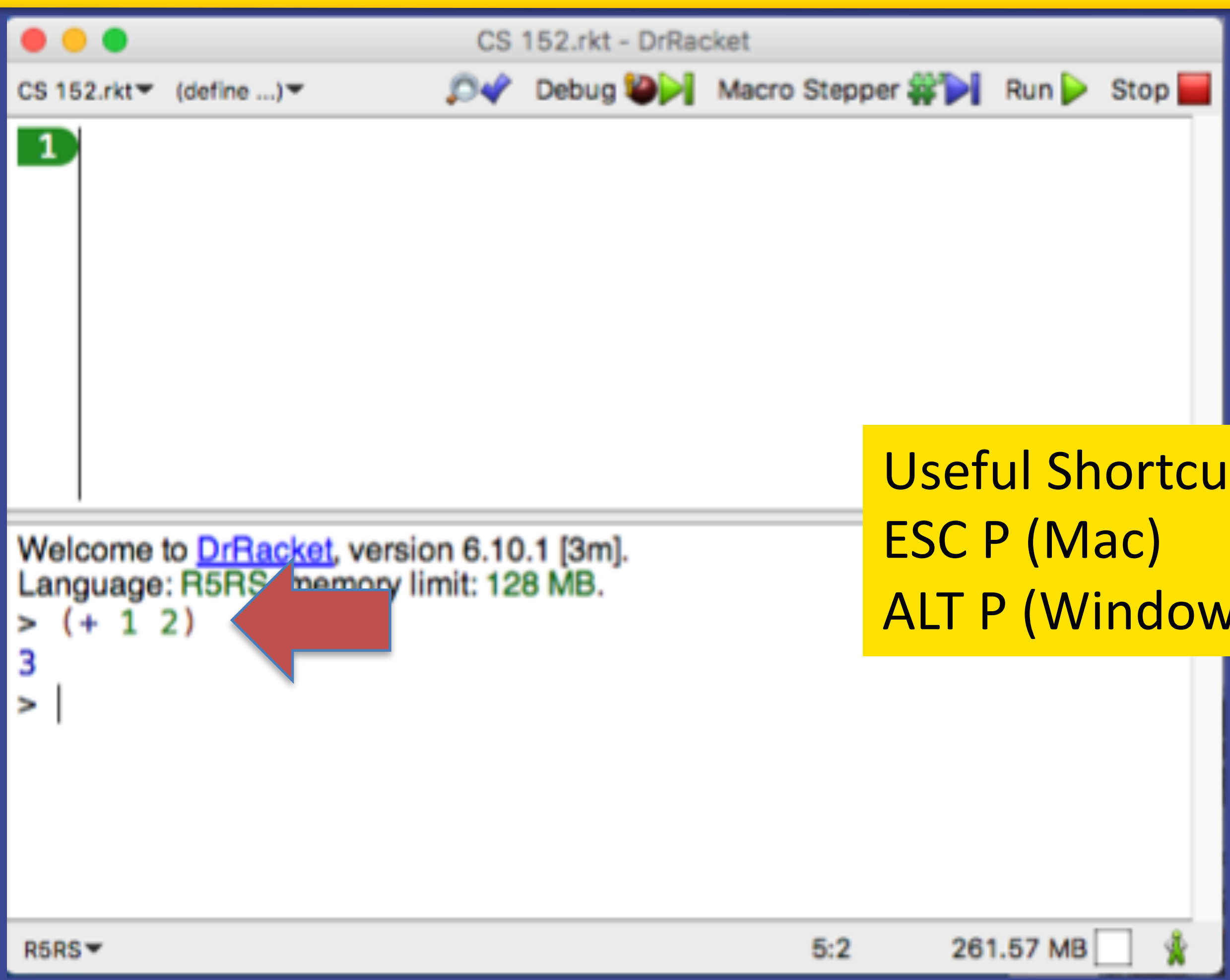


Scheme R5RS in DrRacket



REPL: Read Eval Print Loop

Scheme R5RS in DrRacket



Useful Shortcut:
ESC P (Mac)
ALT P (Windows)

The Elements of Scheme

- All programs and data in Scheme are considered expressions
- Two types of expressions:
  - Atoms: literal constants (characters, booleans, numbers, strings) and identifiers (symbols)
  - Parenthesized expression (list): a sequence of zero or more expressions separated by spaces and surrounded by parentheses

The Elements of Scheme

Two types of expressions:

- Atoms:
  - 3.3 (number)
  - #f (boolean)
  - #\h (character h)
  - "Hello World!" (string)
  - 'grade (symbol)

- Parenthesized expressions
  - A function application
  - A special form

The Elements of Scheme

- When parenthesized expressions are viewed as data, they are called lists

Evaluation Rule for Atoms

- Evaluation rule: the meaning of a Scheme expression
- Atomic literals (characters, booleans, numbers, strings) evaluate to themselves

Evaluation Rule: Symbols

- Symbols other than keywords are treated as identifiers that are looked up in the current symbol table and replaced by values found there
- The symbol table associates symbols (identifiers) with values
- To specify a symbol without evaluating it, we use the quote special form

Symbols

```
> grade
 . grade: undefined; cannot reference undefined identifier
> (quote grade)  ; do not evaluate grade
grade
> 'grade ; this is a just shorthand notation for: (quote grade)
grade
> (symbol? 'grade)
#t
```

Symbols

Symbols are case-insensitive:

```
> (eqv? 'Grade 'grade)
#t
```

Binding Symbols to Values

```
> (define grade 90)
> grade
90
> 'grade
grade
```

Evaluation Rule for Lists

A parenthesized expression (list) is evaluated as follows:

- If the first item is a keyword, a special rule is applied to evaluate the rest of the expression. An expression starting with a keyword is called a special form. (define, quote, …)
- Otherwise, it is a function application. Each expression within the parentheses is evaluated recursively. The first expression must evaluate to a function, which is then applied to remaining values (its arguments)

Function Applications

- All expressions must be written in prefix form: (+ 2 3)
- + is a function, and it is applied to the values 2 and 3, to return the value 5
- A function is represented by the first expression in an application: +
- A function call is surrounded by parentheses (+ 2 3)

To Do

- Homework 2: individual homework
- Due September 8