Racket and Plait

CS 350

Dr. Joseph Eremondi

Last updated: June 18, 2024

Programming in CS 350

All coding for this class uses:

- The Racket Programming Language

All coding for this class uses:

- The Racket Programming Language
- The `plait` library for Racket

All coding for this class uses:

- The Racket Programming Language
- The `plait` library for Racket
- The Dr. Racket editor

Racket

What is Racket?

- Lisp-style language

What is Racket?

- Lisp-style language
  - `(((((((Parentheses)))))))`

What is Racket?

- Lisp-style language
  - `(((((((Parentheses)))))))`
- Language for making languages

What is Dr. Racket?

- IDE for Racket

What is Dr. Racket?

- IDE for Racket
  - Syntax highlighting

What is Dr. Racket?

- IDE for Racket
  - Syntax highlighting
  - Other useful features

What is Dr. Racket?

- IDE for Racket
  - Syntax highlighting
  - Other useful features
- Read-Eval-Print-Loop (REPL)

What is Dr. Racket?

- IDE for Racket
  - Syntax highlighting
  - Other useful features
- Read-Eval-Print-Loop (REPL)
  - Feedback when writing code

What is Dr. Racket?

- IDE for Racket
  - Syntax highlighting
  - Other useful features
- Read-Eval-Print-Loop (REPL)
  - Feedback when writing code
  - Can evaluate expressions while you're writing your code

What is Dr. Racket?

- IDE for Racket
  - Syntax highlighting
  - Other useful features
- Read-Eval-Print-Loop (REPL)
  - Feedback when writing code
  - Can evaluate expressions while you’re writing your code
- Other editors are possible

What is Dr. Racket?

- IDE for Racket
  - Syntax highlighting
  - Other useful features
- Read-Eval-Print-Loop (REPL)
  - Feedback when writing code
  - Can evaluate expressions while you're writing your code
- Other editors are possible
  - … but you're on your own if you have problems

What is Dr. Racket?

- IDE for Racket
  - Syntax highlighting
  - Other useful features
- Read-Eval-Print-Loop (REPL)
  - Feedback when writing code
  - Can evaluate expressions while you're writing your code
- Other editors are possible
  - … but you're on your own if you have problems
  - see `https://docs.racket-lang.org/guide/other-editors.html`

Plait

What is Plait?

“PLAI-typed”

What is Plait?

“PLAI-typed”

Language defined in Racket

What is Plait?

“PLAI-typed”

Language defined in Racket

- Racket functions you can call

What is Plait?

“PLAI-typed”

Language defined in Racket

- Racket functions you can call
- Adds syntax to Racket

What is Plait?

“PLAI-typed”

Language defined in Racket

- Racket functions you can call
- Adds syntax to Racket
    - Declaring and pattern matching on data types

What is Plait?

“PLAI-typed”

Language defined in Racket

- Racket functions you can call
- Adds syntax to Racket
  - Declaring and pattern matching on data types
  - Type annotations for functions

What is Plait?

“PLAI-typed”

Language defined in Racket

- Racket functions you can call
- Adds syntax to Racket
  - Declaring and pattern matching on data types
  - Type annotations for functions
- Minimal

What is Plait?

“PLAI-typed”

Language defined in Racket

- Racket functions you can call
- Adds syntax to Racket
  - Declaring and pattern matching on data types
  - Type annotations for functions
- Minimal
  - Has what you need to write programming languages

What is Plait?

“PLAI-typed”

Language defined in Racket

- Racket functions you can call
- Adds syntax to Racket
  - Declaring and pattern matching on data types
  - Type annotations for functions
- Minimal
  - Has what you need to write programming languages
  - Not much else

What is Plait?

“PLAI-typed”

Language defined in Racket

- Racket functions you can call
- Adds syntax to Racket
  - Declaring and pattern matching on data types
  - Type annotations for functions
- Minimal
  - Has what you need to write programming languages
  - Not much else
  - You can do a lot with very little

Plait features:

- Type inference

Plait features:

- Type inference
  - Every expression is typed

Plait features:

- Type inference
  - Every expression is typed
  - Don’t have to write down the types

Plait features:

- Type inference
  - Every expression is typed
  - Don’t have to write down the types
- Algebraic Data Types

Parentheses

- Racket programs are trees called “S-expressions”

Parentheses

- Racket programs are trees called “S-expressions”
- Parentheses give this tree structure

Parentheses

- Racket programs are trees called “S-expressions”
- Parentheses give this tree structure
- Default: parentheses mean function call

Parentheses

- Racket programs are trees called “S-expressions”
- Parentheses give this tree structure
- Default: parentheses mean function call
  - Racket writes `(f x)`, not `f(x)`

Parentheses

- Racket programs are trees called “S-expressions”
- Parentheses give this tree structure
- Default: parentheses mean function call
  - Racket writes `(f x)`, not `f(x)`
- `x` is not the same as `(x)`

Parentheses

- Racket programs are trees called “S-expressions”
- Parentheses give this tree structure
- Default: parentheses mean function call
  - Racket writes `(f x)`, not `f(x)`
- `x` is not the same as `(x)`
  - `x` gets the value of the variable `x`

Parentheses

- Racket programs are trees called “S-expressions”
- Parentheses give this tree structure
- Default: parentheses mean function call
  - Racket writes `(f x)`, not `f(x)`
- `x` is not the same as `(x)`
  - `x` gets the value of the variable `x`
  - `(x)` is calling a function named `x` with zero arguments

Numbers

Numbers

```
(+ 2 3)
```

Numbers

```
(+ 2 3)
(- 10 0.5)
```

Numbers

```
(+ 2 3)
(- 10 0.5)
(* 1/3 2/3)
```

Numbers

```
(+ 2 3)
(- 10 0.5)
(* 1/3 2/3)
(/ 1 10000000000000.0)
```

Numbers

```
(+ 2 3)
(- 10 0.5)
(* 1/3 2/3)
(/ 1 10000000000000.0)
(max 10 20)
```

Numbers

```
(+ 2 3)
(- 10 0.5)
(* 1/3 2/3)
(/ 1 10000000000000.0)
(max 10 20)
(modulo 10 3)
```

Numbers

```
(+ 2 3)
(- 10 0.5)
(* 1/3 2/3)
(/ 1 10000000000000.0)
(max 10 20)
(modulo 10 3)
```

```
5
```

Numbers

```
(+ 2 3)
(- 10 0.5)
(* 1/3 2/3)
(/ 1 10000000000000.0)
(max 10 20)
(modulo 10 3)
```

```
5
9.5
```

Numbers

```
(+ 2 3)
(- 10 0.5)
(* 1/3 2/3)
(/ 1 10000000000000.0)
(max 10 20)
(modulo 10 3)
```

```
5
9.5
2/9
```

Numbers

```
(+ 2 3)
(- 10 0.5)
(* 1/3 2/3)
(/ 1 10000000000000.0)
(max 10 20)
(modulo 10 3)
```

```
5
9.5
2/9
1e-14
```

Numbers

```
(+ 2 3)
(- 10 0.5)
(* 1/3 2/3)
(/ 1 10000000000000.0)
(max 10 20)
(modulo 10 3)
```

```
5
9.5
2/9
1e-14
20
```

Numbers

```
(+ 2 3)
(- 10 0.5)
(* 1/3 2/3)
(/ 1 10000000000000.0)
(max 10 20)
(modulo 10 3)
```

```
5
9.5
2/9
1e-14
20
1
```

Booleans

Booleans

```
(= (+ 2 3) 5)
```

Booleans

```
(= (+ 2 3) 5)
(> (/ 0 1) 1)
```

Booleans

```
(= (+ 2 3) 5)
(> (/ 0 1) 1)
(zero? (- (+ 1 2) (+ 3 0)))
```

Booleans

```
(= (+ 2 3) 5)
(> (/ 0 1) 1)
(zero? (- (+ 1 2) (+ 3 0)))
(and (< 1 2) (> 1 0))
```

Booleans

```
(= (+ 2 3) 5)
(> (/ 0 1) 1)
(zero? (- (+ 1 2) (+ 3 0)))
(and (< 1 2) (> 1 0))
(or (zero? 1) (even? 3))
```

Booleans

```
(= (+ 2 3) 5)
(> (/ 0 1) 1)
(zero? (- (+ 1 2) (+ 3 0)))
(and (< 1 2) (> 1 0))
(or (zero? 1) (even? 3))
```

```
#t
```

Booleans

```
(= (+ 2 3) 5)
(> (/ 0 1) 1)
(zero? (- (+ 1 2) (+ 3 0)))
(and (< 1 2) (> 1 0))
(or (zero? 1) (even? 3))
```

```
#t
#f
```

Booleans

```
(= (+ 2 3) 5)
(> (/ 0 1) 1)
(zero? (- (+ 1 2) (+ 3 0)))
(and (< 1 2) (> 1 0))
(or (zero? 1) (even? 3))
```

```
#t
#f
#t
```

Booleans

```
(= (+ 2 3) 5)
(> (/ 0 1) 1)
(zero? (- (+ 1 2) (+ 3 0)))
(and (< 1 2) (> 1 0))
(or (zero? 1) (even? 3))
```

```
#t
#f
#t
#t
```

Booleans

```
(= (+ 2 3) 5)
(> (/ 0 1) 1)
(zero? (- (+ 1 2) (+ 3 0)))
(and (< 1 2) (> 1 0))
(or (zero? 1) (even? 3))
```

```
#t
#f
#t
#t
#f
```

Conditionals

- Conditionals are **expressions**, not statements

Conditionals

- Conditionals are **expressions**, not statements
- Boolean changes what the expression **is**, not what it does

Conditionals

- Conditionals are **expressions**, not statements
- Boolean changes what the expression **is**, not what it does

Conditionals

- Conditionals are **expressions**, not statements
- Boolean changes what the expression **is**, not what it does

```
(if (< 2 3) "hello" "goodbye")
```

Conditionals

- Conditionals are **expressions**, not statements
- Boolean changes what the expression **is**, not what it does

```
(if (< 2 3) "hello" "goodbye")
(+ 3
```

Conditionals

- Conditionals are **expressions**, not statements
- Boolean changes what the expression **is**, not what it does

```
(if (< 2 3) "hello" "goodbye")
(+ 3
  (if (= 2 (+ 1 1))
```

Conditionals

- Conditionals are **expressions**, not statements
- Boolean changes what the expression **is**, not what it does

```
(if (< 2 3) "hello" "goodbye")
(+ 3
  (if (= 2 (+ 1 1))
      3
```

Conditionals

- Conditionals are **expressions**, not statements
- Boolean changes what the expression **is**, not what it does

```
(if (< 2 3) "hello" "goodbye")
(+ 3
  (if (= 2 (+ 1 1))
      3
      40))
```

Conditionals

- Conditionals are **expressions**, not statements
- Boolean changes what the expression **is**, not what it does

```
(if (< 2 3) "hello" "goodbye")
(+ 3
  (if (= 2 (+ 1 1))
      3
      40))
```

```
"hello"
```

Conditionals

- Conditionals are **expressions**, not statements
- Boolean changes what the expression **is**, not what it does

```
(if (< 2 3) "hello" "goodbye")
(+ 3
  (if (= 2 (+ 1 1))
      3
      40))
```

```
"hello"
6
```

Functions

- Calling a function replaces variable with concrete argument

Functions

- Calling a function replaces variable with concrete argument

Functions

- Calling a function replaces variable with concrete argument

```
(define (addOne [x : Number]) : Number
```

Functions

- Calling a function replaces variable with concrete argument

```
(define (addOne [x : Number]) : Number
  (+ x 1))
```

Functions

- Calling a function replaces variable with concrete argument

```
(define (addOne [x : Number]) : Number
  (+ x 1))

(addOne 10)
```

Functions

- Calling a function replaces variable with concrete argument

```
(define (addOne [x : Number]) : Number
  (+ x 1))

(addOne 10)

(define (isRemainder [x : Number]
```

Functions

- Calling a function replaces variable with concrete argument

```
(define (addOne [x : Number]) : Number
  (+ x 1))

(addOne 10)

(define (isRemainder [x : Number]
                     [y : Number]
```

Functions

- Calling a function replaces variable with concrete argument

```
(define (addOne [x : Number]) : Number
  (+ x 1))

(addOne 10)

(define (isRemainder [x : Number]
                     [y : Number]
                     [remainder : Number])
```

Functions

- Calling a function replaces variable with concrete argument

```
(define (addOne [x : Number]) : Number
  (+ x 1))

(addOne 10)

(define (isRemainder [x : Number]
                     [y : Number]
                     [remainder : Number])
        : Boolean
```

Functions

- Calling a function replaces variable with concrete argument

```
(define (addOne [x : Number]) : Number
  (+ x 1))

(addOne 10)

(define (isRemainder [x : Number]
                     [y : Number]
                     [remainder : Number])
        : Boolean
  (= remainder (modulo x y)))
```

Functions

- Calling a function replaces variable with concrete argument

```
(define (addOne [x : Number]) : Number
  (+ x 1))

(addOne 10)

(define (isRemainder [x : Number]
                     [y : Number]
                     [remainder : Number])
          : Boolean
  (= remainder (modulo x y)))
(isRemainder 10 3 1)
```

Functions

- Calling a function replaces variable with concrete argument

```
(define (addOne [x : Number]) : Number
  (+ x 1))

(addOne 10)

(define (isRemainder [x : Number]
                     [y : Number]
                     [remainder : Number])
        : Boolean
  (= remainder (modulo x y)))
(isRemainder 10 3 1)
(isRemainder 10 4 1)
```

Functions

- Calling a function replaces variable with concrete argument

```
(define (addOne [x : Number]) : Number
  (+ x 1))

(addOne 10)

(define (isRemainder [x : Number]
                     [y : Number]
                     [remainder : Number])
          : Boolean
  (= remainder (modulo x y)))
(isRemainder 10 3 1)
(isRemainder 10 4 1)
```

```
11
```

Functions

- Calling a function replaces variable with concrete argument

```
(define (addOne [x : Number]) : Number
  (+ x 1))

(addOne 10)

(define (isRemainder [x : Number]
                     [y : Number]
                     [remainder : Number])
          : Boolean
  (= remainder (modulo x y)))
(isRemainder 10 3 1)
(isRemainder 10 4 1)
```

```
11
#t
```

Functions

- Calling a function replaces variable with concrete argument

```
(define (addOne [x : Number]) : Number
  (+ x 1))

(addOne 10)

(define (isRemainder [x : Number]
                     [y : Number]
                     [remainder : Number])
        : Boolean
  (= remainder (modulo x y)))
(isRemainder 10 3 1)
(isRemainder 10 4 1)
```

```
11
#t
#f
```

Functions (ctd.)

- General form:

Functions (ctd.)

- General form:

Functions (ctd.)

- General form:

```
(define (functionName
```

Functions (ctd.)

- General form:

```
(define (functionName
          [argName : argType]
```

Functions (ctd.)

- General form:

```
(define (functionName
          [argName : argType]
          ... [argNameN : argTypeN]) : returnType
```

Functions (ctd.)

- General form:

```
(define (functionName
          [argName : argType]
          ... [argNameN : argTypeN]) : returnType
  functionBody)
```

- Later in the course we'll see another way of defining functions

Functional Thinking: Lists And Recursion

What Is Functional Programming?

- Functions in our program correspond to functions in math

What Is Functional Programming?

- Functions in our program correspond to functions in math
  - Mapping from inputs to outputs

What Is Functional Programming?

- Functions in our program correspond to functions in math
  - Mapping from inputs to outputs
  - Same inputs always produce the same outputs

What Is Functional Programming?

- Functions in our program correspond to functions in math
  - Mapping from inputs to outputs
  - Same inputs always produce the same outputs
- Talk about what programs **are**, not what programs do

What Is Functional Programming?

- Functions in our program correspond to functions in math
  - Mapping from inputs to outputs
  - Same inputs always produce the same outputs
- Talk about what programs **are**, not what programs do
- Instead of changing variable values

What Is Functional Programming?

- Functions in our program correspond to functions in math
  - Mapping from inputs to outputs
  - Same inputs always produce the same outputs
- Talk about what programs **are**, not what programs do
- Instead of changing variable values
  - We call functions with different arguments

What Is Functional Programming?

- Functions in our program correspond to functions in math
  - Mapping from inputs to outputs
  - Same inputs always produce the same outputs
- Talk about what programs **are**, not what programs do
- Instead of changing variable values
  - We call functions with different arguments
- Instead of changing data structures

What Is Functional Programming?

- Functions in our program correspond to functions in math
  - Mapping from inputs to outputs
  - Same inputs always produce the same outputs
- Talk about what programs **are**, not what programs do
- Instead of changing variable values
  - We call functions with different arguments
- Instead of changing data structures
  - We decompose them, copy the parts, and reassemble them in new ways

What Is Functional Programming?

- Functions in our program correspond to functions in math
  - Mapping from inputs to outputs
  - Same inputs always produce the same outputs
- Talk about what programs **are**, not what programs do
- Instead of changing variable values
  - We call functions with different arguments
- Instead of changing data structures
  - We decompose them, copy the parts, and reassemble them in new ways
  - Copying is implemented with pointers

What Is Functional Programming?

- Functions in our program correspond to functions in math
  - Mapping from inputs to outputs
  - Same inputs always produce the same outputs
- Talk about what programs **are**, not what programs do
- Instead of changing variable values
  - We call functions with different arguments
- Instead of changing data structures
  - We decompose them, copy the parts, and reassemble them in new ways
  - Copying is implemented with pointers
    - Fast, memory efficient

How to design functional programs

5 Step method:

How to design functional programs

5 Step method:

1. Determine the representation of inputs and outputs

How to design functional programs

5 Step method:

1. Determine the representation of inputs and outputs
2. Write examples/tests

How to design functional programs

5 Step method:

1. Determine the representation of inputs and outputs
2. Write examples/tests
3. Create a **template** of the function

How to design functional programs

5 Step method:

1. Determine the representation of inputs and outputs
2. Write examples/tests
3. Create a **template** of the function
    3.1 Depends on input/output types

How to design functional programs

5 Step method:

1. Determine the representation of inputs and outputs
2. Write examples/tests
3. Create a **template** of the function
    3.1 Depends on input/output types
    3.2 Covers all cases

How to design functional programs

5 Step method:

1. Determine the representation of inputs and outputs
2. Write examples/tests
3. Create a **template** of the function
    3.1 Depends on input/output types
    3.2 Covers all cases
    3.3 Possibly extracts fields, recursive calls, etc.

How to design functional programs

5 Step method:

1. Determine the representation of inputs and outputs
2. Write examples/tests
3. Create a **template** of the function
    3.1 Depends on input/output types
    3.2 Covers all cases
    3.3 Possibly extracts fields, recursive calls, etc.
4. Fill in the holes in the template

How to design functional programs

5 Step method:

1. Determine the representation of inputs and outputs
2. Write examples/tests
3. Create a **template** of the function
   - 3.1 Depends on input/output types
   - 3.2 Covers all cases
   - 3.3 Possibly extracts fields, recursive calls, etc.
4. Fill in the holes in the template
5. Run tests

How to design functional programs

5 Step method:

1. Determine the representation of inputs and outputs
2. Write examples/tests
3. Create a **template** of the function
    3.1 Depends on input/output types
    3.2 Covers all cases
    3.3 Possibly extracts fields, recursive calls, etc.
4. Fill in the holes in the template
5. Run tests

Further reference:
`http://htdp.org`, Matthew Flatt's Notes (URCourses)