Functional Programming 1: Recursion and Immutable Data

CS 350

Dr. Joseph Eremondi

Last updated: June 21, 2024 July 2,

Overview

- Topic: Functional Programming in Racket and plait

Overview

- Topic: Functional Programming in Racket and plait
- Required Reading:

Overview

- Topic: Functional Programming in Racket and plait
- Required Reading:
    - Plait tutorial (URCourses)

Overview

- Topic: Functional Programming in Racket and plait
- Required Reading:
  - Plait tutorial (URCourses)
- Optional Reference

Overview

- Topic: Functional Programming in Racket and plait
- Required Reading:
  - Plait tutorial (URCourses)
- Optional Reference
  - Plait videos, HtDP videos

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

Numbers

Numbers

```
(+ 2 7)
```

Numbers

```
(+ 2 7)
```

Numbers

```
(+ 2 7)
(- 10 0.5)
```

9

Numbers

```
(+ 2 7)
(- 10 0.5)
```

```
9
9.5
```

Numbers

```
(+ 2 7)
(- 10 0.5)
(* 1/3 2/3)
```

```
9
9.5
```

Numbers

```
(+ 2 7)
(- 10 0.5)
(* 1/3 2/3)
```

```
9
9.5
2/9
```

Numbers

```
(+ 2 7)
(- 10 0.5)
(* 1/3 2/3)
(/ 1 1000000000000.0)
```

```
9
9.5
2/9
```

Numbers

```
(+ 2 7)
(- 10 0.5)
(* 1/3 2/3)
(/ 1 1000000000000.0)
```

```
9
9.5
2/9
1e-12
```

Numbers

```
(+ 2 7)
(- 10 0.5)
(* 1/3 2/3)
(/ 1 1000000000000.0)
(max 10 20)
```

```
9
9.5
2/9
1e-12
```

Numbers

```
(+ 2 7)
(- 10 0.5)
(* 1/3 2/3)
(/ 1 1000000000000.0)
(max 10 20)
```

```
9
9.5
2/9
1e-12
20
```

Numbers

```
(+ 2 7)
(- 10 0.5)
(* 1/3 2/3)
(/ 1 1000000000000.0)
(max 10 20)
(modulo 10 3)
```

```
9
9.5
2/9
1e-12
20
```

Numbers

```
(+ 2 7)
(- 10 0.5)
(* 1/3 2/3)
(/ 1 1000000000000.0)
(max 10 20)
(modulo 10 3)
```

```
9
9.5
2/9
1e-12
20
1
```

Booleans

Booleans

Booleans

Booleans

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

Booleans

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

Booleans

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

```
#t
```

Booleans

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

```
#t
#f
```

Booleans

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

```
#t
#f
```

Booleans

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

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

Booleans

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

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

Booleans

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

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

Booleans

```scheme
(= (+ 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")
(+ 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"
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
  (+ x 1))
(addOne 10)
```

Functions

- Calling a function replaces variable with concrete argument

```
(define (addOne [x : Number]) : Number
  (+ x 1))
(addOne 10)
```

```
11
```

Functions

Functions

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

Functions

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

```
#t
#f
```

Functions (ctd.)

- General form:

Functions (ctd.)

- General form:

Functions (ctd.)

- General form:

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

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

Symbols

- Special type in Racket

Symbols

- Special type in Racket
- Written with single quote `'a`, `'hello`, `'foo`

Symbols

- Special type in Racket
- Written with single quote `'a`, `'hello`, `'foo`
- Like strings, but you don't ever traverse/concatenate/look inside

Symbols

- Special type in Racket
- Written with single quote `'a`, `'hello`, `'foo`
- Like strings, but you don't ever traverse/concatenate/look inside
- Only relevant operation is comparison

Symbols

- Special type in Racket
- Written with single quote `'a`, `'hello`, `'foo`
- Like strings, but you don't ever traverse/concatenate/look inside
- Only relevant operation is comparison
  - `(symbol=?  'a 'b)`

Symbols

- Special type in Racket
- Written with single quote `'a`, `'hello`, `'foo`
- Like strings, but you don't ever traverse/concatenate/look inside
- Only relevant operation is comparison
  - `(symbol=?  'a 'b)`
  - Compares pointers, so very fast

Intermediate definitions

- Can still define variables

Intermediate definitions

- Can still define variables
  - Once they’re given a value, never changes

Intermediate definitions

- Can still define variables
  - Once they’re given a value, never changes
  - Allows re-use

Intermediate definitions

- Can still define variables
  - Once they're given a value, never changes
  - Allows re-use
    - Only evaluated once, can use multiple times

Intermediate definitions

- Can still define variables
  - Once they're given a value, never changes
  - Allows re-use
    - Only evaluated once, can use multiple times

Intermediate definitions

- Can still define variables
  - Once they're given a value, never changes
  - Allows re-use
    - Only evaluated once, can use multiple times

```
(define (squaredSum [x : Number]
                    [y : Number]) : Number
  (let ([xy (+ x y)])
    (* xy xy)))
(squaredSum 1 2)
```

Intermediate definitions

- Can still define variables
  - Once they're given a value, never changes
  - Allows re-use
    - Only evaluated once, can use multiple times

```
(define (squaredSum [x : Number]
                    [y : Number]) : Number
  (let ([xy (+ x y)])
    (* xy xy)))
(squaredSum 1 2)
```

```
9
```

Alternate Versions of Let

- `let*` : multiple definitions, later ones can refer to earlier ones

Alternate Versions of Let

- `let*`: multiple definitions, later ones can refer to earlier ones
- 99% of the time this is what you want to use

Alternate Versions of Let

- `let*`: multiple definitions, later ones can refer to earlier ones
- 99% of the time this is what you want to use

Alternate Versions of Let

- `let*` : multiple definitions, later ones can refer to earlier ones
- 99% of the time this is what you want to use

```
(let* ([x (+ 2 3)]
       [y (* x x)])
  (* y y))
```

- `letrec` : multiple definitions that can all refer to each other

Alternate Versions of Let

- `let*` : multiple definitions, later ones can refer to earlier ones
- 99% of the time this is what you want to use

```
(let* ([x (+ 2 3)]
       [y (* x x)])
  (* y y))
```

- `letrec` : multiple definitions that can all refer to each other
    - We'll see this later when we learn about lambdas

Functional Thinking 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

Advantages of Functional Programming

- All program state is **explicit**

Advantages of Functional Programming

- All program state is **explicit**
  - Easy to tell exactly what a function can change

Advantages of Functional Programming

- All program state is **explicit**
  - Easy to tell exactly what a function can change
  - No shared state between components

Advantages of Functional Programming

- All program state is **explicit**
    - Easy to tell exactly what a function can change
    - No shared state between components
        - Other function can't change value without realizing

Advantages of Functional Programming

- All program state is **explicit**
    - Easy to tell exactly what a function can change
    - No shared state between components
        - Other function can't change value without realizing
        - No data races for threading

Advantages of Functional Programming

- All program state is **explicit**
  - Easy to tell exactly what a function can change
  - No shared state between components
    - Other function can’t change value without realizing
    - No data races for threading
- Programming is **declarative**

Advantages of Functional Programming

- All program state is **explicit**
  - Easy to tell exactly what a function can change
  - No shared state between components
    - Other function can’t change value without realizing
    - No data races for threading
- Programming is **declarative**
  - Structure of the problem guides structure of the solution

Advantages of Functional Programming

- All program state is **explicit**
    - Easy to tell exactly what a function can change
    - No shared state between components
        - Other function can't change value without realizing
        - No data races for threading
- Programming is **declarative**
    - Structure of the problem guides structure of the solution
- Equational reasoning

Advantages of Functional Programming

- All program state is **explicit**
  - Easy to tell exactly what a function can change
  - No shared state between components
    - Other function can't change value without realizing
    - No data races for threading
- Programming is **declarative**
  - Structure of the problem guides structure of the solution
- Equational reasoning
  - In imperative languages, equals sign = is a LIE

Advantages of Functional Programming

- All program state is **explicit**
  - Easy to tell exactly what a function can change
  - No shared state between components
    - Other function can't change value without realizing
    - No data races for threading
- Programming is **declarative**
  - Structure of the problem guides structure of the solution
- Equational reasoning
  - In imperative languages, equals sign = is a LIE
    - Can write `x = 3; x = 4;`, but `3 != 4`

Advantages of Functional Programming

- All program state is **explicit**
  - Easy to tell exactly what a function can change
  - No shared state between components
    - Other function can't change value without realizing
    - No data races for threading
- Programming is **declarative**
  - Structure of the problem guides structure of the solution
- Equational reasoning
  - In imperative languages, equals sign = is a LIE
    - Can write `x = 3; x = 4;`, but `3 != 4`
  - If have `(define (f x) body)`, then for all `y`, `(f y)` and `body` are interchangable

Advantages of Functional Programming

- All program state is **explicit**
  - Easy to tell exactly what a function can change
  - No shared state between components
    - Other function can't change value without realizing
    - No data races for threading
- Programming is **declarative**
  - Structure of the problem guides structure of the solution
- Equational reasoning
  - In imperative languages, equals sign = is a LIE
    - Can write `x = 3; x = 4;`, but `3 != 4`
  - If have `(define (f x) body)`, then for all `y`, `(f y)` and `body` are interchangable
    - after replace `x` with `y` in `body`

Advantages of Functional Programming

- All program state is **explicit**
  - Easy to tell exactly what a function can change
  - No shared state between components
    - Other function can't change value without realizing
    - No data races for threading
- Programming is **declarative**
  - Structure of the problem guides structure of the solution
- Equational reasoning
  - In imperative languages, equals sign = is a LIE
    - Can write `x = 3; x = 4;`, but `3 != 4`
  - If have `(define (f x) body)`, then for all `y`, `(f y)` and `body` are interchangable
    - after replace `x` with `y` in `body`
    - Easier to tell if your program is correct

Advantages of Functional Programming

- All program state is **explicit**
  - Easy to tell exactly what a function can change
  - No shared state between components
    - Other function can't change value without realizing
    - No data races for threading
- Programming is **declarative**
  - Structure of the problem guides structure of the solution
- Equational reasoning
  - In imperative languages, equals sign = is a LIE
    - Can write `x = 3; x = 4;`, but `3 != 4`
  - If have `(define (f x) body)`, then for all `y`, `(f y)` and `body` are interchangable
    - after replace `x` with `y` in `body`
    - Easier to tell if your program is correct
    - Some optimizations easier

Disadvantages of Functional Programming

- None?

Disadvantages of Functional Programming

- None?
- Sometimes slower

Disadvantages of Functional Programming

- None?
- Sometimes slower
  - Very hard to do without Garbage Collection

Disadvantages of Functional Programming

- None?
- Sometimes slower
  - Very hard to do without Garbage Collection
    - e.g. see Closures in Rust

Disadvantages of Functional Programming

- None?
- Sometimes slower
    - Very hard to do without Garbage Collection
        - e.g. see Closures in Rust
    - Sometimes faster because you need fewer safety checks in your code

Disadvantages of Functional Programming

- None?
- Sometimes slower
  - Very hard to do without Garbage Collection
    - e.g. see Closures in Rust
  - Sometimes faster because you need fewer safety checks in your code
- Farther from what the CPU is actually doing

Disadvantages of Functional Programming

- None?
- Sometimes slower
  - Very hard to do without Garbage Collection
    - e.g. see Closures in Rust
  - Sometimes faster because you need fewer safety checks in your code
- Farther from what the CPU is actually doing
- Some algorithms are more concise with mutation

Disadvantages of Functional Programming

- None?
- Sometimes slower
  - Very hard to do without Garbage Collection
    - e.g. see Closures in Rust
  - Sometimes faster because you need fewer safety checks in your code
- Farther from what the CPU is actually doing
- Some algorithms are more concise with mutation
  - But lots aren't

How to design functional programs

5 Step method:

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** and tests

How to design functional programs

5 Step method:

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

How to design functional programs

5 Step method:

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

How to design functional programs

5 Step method:

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

Factorial - Representation

- $n! = 1 \times 2 \times \ldots n$

Factorial - Representation

- $n! = 1 \times 2 \times \ldots n$
- Takes in a (natural) number, outputs a number

Factorial - Representation

- $n! = 1 \times 2 \times \ldots n$
- Takes in a (natural) number, outputs a number

Factorial - Representation

- $n! = 1 \times 2 \times \ldots n$
- Takes in a (natural) number, outputs a number

```
(define (factorial [n : Number]) : Number
  (error 'factorial "TODO"))
```

Factorial - Examples

Factorial - Examples

Factorial - Examples

Factorial - Examples

```
(test (factorial 0) 1 )
```

Factorial - Examples

```
(test (factorial 0) 1 )
```

Factorial - Examples

```
(test (factorial 0) 1 )
(test (factorial 1) 1 )
```

Factorial - Examples

```
(test (factorial 0) 1 )
(test (factorial 1) 1 )
```

Factorial - Examples

```
(test (factorial 0) 1 )
(test (factorial 1) 1 )
(test (factorial 2) 2 )
```

Factorial - Examples

```
(test (factorial 0) 1 )
(test (factorial 1) 1 )
(test (factorial 2) 2 )
```

Factorial - Examples

```
(test (factorial 0) 1 )
(test (factorial 1) 1 )
(test (factorial 2) 2 )
(test (factorial 3) 6 )
```

Factorial - Examples

```
(test (factorial 0) 1 )
(test (factorial 1) 1 )
(test (factorial 2) 2 )
(test (factorial 3) 6 )
```

Factorial - Examples

```
(test (factorial 0) 1 )
(test (factorial 1) 1 )
(test (factorial 2) 2 )
(test (factorial 3) 6 )
(test (factorial 4) 24 )
```

Factorial - Examples

```
(test (factorial 0) 1 )
(test (factorial 1) 1 )
(test (factorial 2) 2 )
(test (factorial 3) 6 )
(test (factorial 4) 24 )
```

Factorial - Examples

```
(test (factorial 0) 1 )
(test (factorial 1) 1 )
(test (factorial 2) 2 )
(test (factorial 3) 6 )
(test (factorial 4) 24 )
(test (factorial 5) 120 )
```

- Notice the pattern?

Factorial - Template

- A natural number is either

Factorial - Template

- A natural number is either
  - Zero

Factorial - Template

- A natural number is either
  - Zero
  - One more than some other number

Factorial - Template

- A natural number is either
  - Zero
  - One more than some other number
    - We call this the “successor”, written “S” or “suc”

Factorial - Template

- A natural number is either
  - Zero
  - One more than some other number
    - We call this the “successor”, written “S” or “suc”
    - Probably want to use this in the solution

Factorial - Template

- A natural number is either
  - Zero
  - One more than some other number
    - We call this the “successor”, written “S” or “suc”
    - Probably want to use this in the solution

Factorial - Template

- A natural number is either
  - Zero
  - One more than some other number
    - We call this the "successor", written "S" or "suc"
    - Probably want to use this in the solution

```
(define (factorial [n : Number]) : Number
  (if (zero? n)
      (error 'zero "TODO")
      (let ([n-1 (- n 1)])
        (error 'suc "TODO"))
      ))
```

Factorial - Recursion

- Divide problem into base case and recursive cases

Factorial - Recursion

- Divide problem into base case and recursive cases
- Can use recursive calls to smaller arguments

Factorial - Recursion

- Divide problem into base case and recursive cases
- Can use recursive calls to smaller arguments
- Build up solution in terms of solutions to smaller problems

Factorial - Recursion

- Divide problem into base case and recursive cases
- Can use recursive calls to smaller arguments
- Build up solution in terms of solutions to smaller problems

Factorial - Recursion

- Divide problem into base case and recursive cases
- Can use recursive calls to smaller arguments
- Build up solution in terms of solutions to smaller problems

```
(define (factorial [n : Number]) : Number
  (if (zero? n)
      (error 'zero "TODO")
      (let*
          ([n-1 (- n 1)]
           [fn-1 (factorial n-1)])
          (error 'suc "TODO"))
      ))
```

Factorial - Filling holes

- Example gives the base case for `0`

Factorial - Filling holes

- Example gives the base case for `0`
- Notice the pattern

Factorial - Filling holes

- Example gives the base case for `0`
- Notice the pattern
  - Multiplying the first n numbers is the same as n times the first n-1 numbers

Factorial - Filling holes

- Example gives the base case for `0`
- Notice the pattern
  - Multiplying the first n numbers is the same as n times the first n-1 numbers
  - We get that from our recursive call

Factorial - Filling holes

- Example gives the base case for `0`
- Notice the pattern
  - Multiplying the first n numbers is the same as n times the first n-1 numbers
  - We get that from our recursive call

Factorial - Filling holes

- Example gives the base case for `0`
- Notice the pattern
  - Multiplying the first n numbers is the same as n times the first n-1 numbers
  - We get that from our recursive call

```
(define (factorial [n : Number]) : Number
  (if (zero? n)
      1
      (let*
          ([n-1 (- n 1)]
           [fn-1 (factorial n-1)])
          (* n fn-1))
      ))
```

Run Tests

Run Tests

```
(test (factorial 0) 1 )
(test (factorial 5) 120 )
```

Run Tests

```
(test (factorial 0) 1 )
(test (factorial 5) 120 )
```

```
good (factorial 0) at line 11
  expected: 1
  given: 1

good (factorial 5) at line 12
  expected: 120
  given: 120
```

Trust the Natural Recursion

- The shape of the data guided the shape of the solution

Trust the Natural Recursion

- The shape of the data guided the shape of the solution
    - Zero had no sub-data, so there were no recursive calls

Trust the Natural Recursion

- The shape of the data guided the shape of the solution
    - Zero had no sub-data, so there were no recursive calls
    - *suc n* has one sub-value, namely *n*

Trust the Natural Recursion

- The shape of the data guided the shape of the solution
    - Zero had no sub-data, so there were no recursive calls
    - *suc n* has one sub-value, namely *n*
        - One recursive call

Trust the Natural Recursion

- The shape of the data guided the shape of the solution
  - Zero had no sub-data, so there were no recursive calls
  - *suc n* has one sub-value, namely *n*
    - One recursive call
- Note: types not quite precise enough

Trust the Natural Recursion

- The shape of the data guided the shape of the solution
  - Zero had no sub-data, so there were no recursive calls
  - *suc n* has one sub-value, namely *n*
    - One recursive call
- Note: types not quite precise enough
  - e.g. `(factorial -1)` or `(factorial 1/2)` loop forever

Trust the Natural Recursion

- The shape of the data guided the shape of the solution
    - Zero had no sub-data, so there were no recursive calls
    - *suc n* has one sub-value, namely *n*
        - One recursive call
- Note: types not quite precise enough
    - e.g. `(factorial -1)` or `(factorial 1/2)` loop forever
- Precondition: argument is a non-negative whole number

Trust the Natural Recursion

- The shape of the data guided the shape of the solution
  - Zero had no sub-data, so there were no recursive calls
  - *suc n* has one sub-value, namely *n*
    - One recursive call
- Note: types not quite precise enough
  - e.g. `(factorial -1)` or `(factorial 1/2)` loop forever
- Precondition: argument is a non-negative whole number
  - Can't express this in the code, so write in the comments

Trust the Natural Recursion

- The shape of the data guided the shape of the solution
  - Zero had no sub-data, so there were no recursive calls
  - *suc n* has one sub-value, namely *n*
    - One recursive call
- Note: types not quite precise enough
  - e.g. `(factorial -1)` or `(factorial 1/2)` loop forever
- Precondition: argument is a non-negative whole number
  - Can't express this in the code, so write in the comments
  - Aside: I research languages where you *can* express this with types

Another Example: Exponentiation

- Live coding in Dr. Racket

Unbounded Data: Lists

Functional Linked Lists

- Every linked list is one of:

Functional Linked Lists

- Every linked list is one of:
  - Empty (sometimes called `nil` or `null`)

Functional Linked Lists

- Every linked list is one of:
  - Empty (sometimes called `nil` or `null`)
  - An element appended to the beginning of another list

Functional Linked Lists

- Every linked list is one of:
    - Empty (sometimes called `nil` or `null`)
    - An element appended to the beginning of another list
- We call the operation of appending an element to a list `cons`

Functional Linked Lists

- Every linked list is one of:
  - Empty (sometimes called `nil` or `null`)
  - An element appended to the beginning of another list
- We call the operation of appending an element to a list `cons`
  - Historical name, goes back to LISP days

Functional Linked Lists

- Every linked list is one of:
  - Empty (sometimes called `nil` or `null`)
  - An element appended to the beginning of another list
- We call the operation of appending an element to a list `cons`
  - Historical name, goes back to LISP days
- Cons does **not** change its input

Functional Linked Lists

- Every linked list is one of:
  - Empty (sometimes called `nil` or `null`)
  - An element appended to the beginning of another list
- We call the operation of appending an element to a list `cons`
  - Historical name, goes back to LISP days
- Cons does **not** change its input
  - Creates a new list whose tail is the old list

Lists in Racket

- Multiple ways to write lists

Lists in Racket

- Multiple ways to write lists
- `'()` is the empty list, can also write `empty`

Lists in Racket

- Multiple ways to write lists
- `'()` is the empty list, can also write `empty`
- Extending lists: `(cons h t)` creates list with element `h` appended to list `t`

Lists in Racket

- Multiple ways to write lists
- `'()` is the empty list, can also write `empty`
- Extending lists: `(cons h t)` creates list with element `h` appended to list `t`
  - `h` and `t` for `head` and `tail`

Lists in Racket

- Multiple ways to write lists
- `'()` is the empty list, can also write `empty`
- Extending lists: `(cons h t)` creates list with element `h` appended to list `t`
    - `h` and `t` for `head` and `tail`
- List literals, can write `(list 1 2 3 4)` or `'(1 2 3 4)`

Lists in Racket

- Multiple ways to write lists
- `'()` is the empty list, can also write `empty`
- Extending lists: `(cons h t)` creates list with element `h` appended to list `t`
    - `h` and `t` for `head` and `tail`
- List literals, can write `(list 1 2 3 4)` or `'(1 2 3 4)`
    - Shorthand for:

Lists in Racket

- Multiple ways to write lists
- `'()` is the empty list, can also write `empty`
- Extending lists: `(cons h t)` creates list with element `h` appended to list `t`
    - `h` and `t` for `head` and `tail`
- List literals, can write `(list 1 2 3 4)` or `'(1 2 3 4)`
    - Shorthand for:
    - `(cons 1 (cons 2 (cons 3 (cons 4 '()))))`

Lists in Racket

- Multiple ways to write lists
- `'()` is the empty list, can also write `empty`
- Extending lists: `(cons h t)` creates list with element `h` appended to list `t`
    - `h` and `t` for `head` and `tail`
- List literals, can write `(list 1 2 3 4)` or `'(1 2 3 4)`
    - Shorthand for:
    - `(cons 1 (cons 2 (cons 3 (cons 4 '()))))`
- Lots more helper functions, see the documentation

Template for Lists

- Two cases: list is empty or cons

Template for Lists

- Two cases: list is empty or cons
- Make a recursive call on tail of cons case

Template for Lists

- Two cases: list is empty or cons
- Make a recursive call on tail of cons case

Template for Lists

- Two cases: list is empty or cons
- Make a recursive call on tail of cons case

```
(define (list-template
         [xs : (Listof Number)])
  (if (empty? xs)
      (error 'nil "TODO")
      (let ([h (first xs)]
            [t (rest xs)]
            [tRet (list-template t)])
        (error 'cons "TODO"))
      ))
```

Example: Sum

Example: Sum

```
(define (sum [xs : (Listof Number)])
  : Number
  (if (empty? xs)
      0
      (let* ([h (first xs)]
             [t (rest xs)]
             [tRet (sum t)])
        (+ h tRet))))
 (sum '())
 (sum '(1 2 3))
```

Example: Sum

```
(define (sum [xs : (Listof Number)])
  : Number
  (if (empty? xs)
      0
      (let* ([h (first xs)]
             [t (rest xs)]
             [tRet (sum t)])
        (+ h tRet))))
 (sum '())
 (sum '(1 2 3))
```

```
0
6
```

Pattern Matching: Motivation

- Recursive case: used "getter" function to get the sub-data in the recursive case

Pattern Matching: Motivation

- Recursive case: used “getter” function to get the sub-data in the recursive case
  - `(- x 1)` for numbers

Pattern Matching: Motivation

- Recursive case: used "getter" function to get the sub-data in the recursive case
  - `(- x 1)` for numbers
  - `first` and `rest` for lists

Pattern Matching: Motivation

- Recursive case: used “getter” function to get the sub-data in the recursive case
  - `(- x 1)` for numbers
  - `first` and `rest` for lists
- Always want to have the sub-parts available

Pattern Matching: Motivation

- Recursive case: used “getter” function to get the sub-data in the recursive case
  - `(- x 1)` for numbers
  - `first` and `rest` for lists
- Always want to have the sub-parts available
- Don’t want to apply getters on the wrong data

Pattern Matching: Motivation

- Recursive case: used “getter” function to get the sub-data in the recursive case
  - `(- x 1)` for numbers
  - `first` and `rest` for lists
- Always want to have the sub-parts available
- Don’t want to apply getters on the wrong data
  - e.g. `first ’()` will raise an error

Pattern Matching:

Example: Generating a Modified List

- e.g. Increment each number in a list

Example: Generating a Modified List

- e.g. Increment each number in a list
  - Uses pattern matching

Example: Generating a Modified List

- e.g. Increment each number in a list
  - Uses pattern matching
  - Shows how to create lists recursively

Example: Generating a Modified List

- e.g. Increment each number in a list
    - Uses pattern matching
    - Shows how to create lists recursively

Example: Generating a Modified List

- e.g. Increment each number in a list
  - Uses pattern matching
  - Shows how to create lists recursively

```
(define (increment [xs : (Listof Number)])
        : (Listof Number)
  (type-case (Listof Number) xs
    [empty
       empty]
    [(cons h t)
       (cons (+ h 1) (increment t))]))
(increment '(2 3 4))
```

Example: Generating a Modified List

- e.g. Increment each number in a list
  - Uses pattern matching
  - Shows how to create lists recursively

```
(define (increment [xs : (Listof Number)])
        : (Listof Number)
  (type-case (Listof Number) xs
    [empty
       empty]
    [(cons h t)
       (cons (+ h 1) (increment t))]))
(increment '(2 3 4))
```

```
  '(3 4 5)
```

Parametric Polymorphism

- Lists are a **parameterized type**

Parametric Polymorphism

- Lists are a **parameterized type**
  - Only need to define once for the different element types

Parametric Polymorphism

- Lists are a **parameterized type**
  - Only need to define once for the different element types
- Many list functions are **polymorphic**

Parametric Polymorphism

- Lists are a **parameterized type**
  - Only need to define once for the different element types
- Many list functions are **polymorphic**
  - Work regardless of what type of elements there are

Parametric Polymorphism

- Lists are a **parameterized type**
  - Only need to define once for the different element types
- Many list functions are **polymorphic**
  - Work regardless of what type of elements there are
  - Types contain **type variables**, denoted with single quote `'x`

Parametric Polymorphism

- Lists are a **parameterized type**
  - Only need to define once for the different element types
- Many list functions are **polymorphic**
  - Work regardless of what type of elements there are
  - Types contain **type variables**, denoted with single quote `'x`
    - Like symbols

Parametric Polymorphism

- Lists are a **parameterized type**
  - Only need to define once for the different element types
- Many list functions are **polymorphic**
  - Work regardless of what type of elements there are
  - Types contain **type variables**, denoted with single quote `'x`
    - Like symbols
  - Plait type inference figures out solutions for type variables when you call a function

Parametric Polymorphism

- Lists are a **parameterized type**
  - Only need to define once for the different element types
- Many list functions are **polymorphic**
  - Work regardless of what type of elements there are
  - Types contain **type variables**, denoted with single quote `'x`
    - Like symbols
  - Plait type inference figures out solutions for type variables when you call a function
  - E.g. `first : ((Listof 'a) -> 'a)`

Parametric Polymorphism

- Lists are a **parameterized type**
  - Only need to define once for the different element types
- Many list functions are **polymorphic**
  - Work regardless of what type of elements there are
  - Types contain **type variables**, denoted with single quote `'x`
    - Like symbols
  - Plait type inference figures out solutions for type variables when you call a function
  - E.g. `first : ((Listof 'a) -> 'a)`
    - Input is list whose elements are some type `'a`

Parametric Polymorphism

- Lists are a **parameterized type**
  - Only need to define once for the different element types
- Many list functions are **polymorphic**
  - Work regardless of what type of elements there are
  - Types contain **type variables**, denoted with single quote `'x`
    - Like symbols
  - Plait type inference figures out solutions for type variables when you call a function
  - E.g. `first : ((Listof 'a) -> 'a)`
    - Input is list whose elements are some type `'a`
    - Output has type `'a`

Parametric Polymorphism

- Lists are a **parameterized type**
  - Only need to define once for the different element types
- Many list functions are **polymorphic**
  - Work regardless of what type of elements there are
  - Types contain **type variables**, denoted with single quote `'x`
    - Like symbols
  - Plait type inference figures out solutions for type variables when you call a function
  - E.g. `first : ((Listof 'a) -> 'a)`
    - Input is list whose elements are some type `'a`
    - Output has type `'a`
    - e.g. `first '(1 2 3)` is a `Number`, but `first '(#t #f #t)` is a Boolean

Parametric Polymorphism

- Lists are a **parameterized type**
  - Only need to define once for the different element types
- Many list functions are **polymorphic**
  - Work regardless of what type of elements there are
  - Types contain **type variables**, denoted with single quote `'x`
    - Like symbols
  - Plait type inference figures out solutions for type variables when you call a function
  - E.g. `first : ((Listof 'a) -> 'a)`
    - Input is list whose elements are some type `'a`
    - Output has type `'a`
    - e.g. `first '(1 2 3)` is a `Number`, but `first '(#t #f #t)` is a Boolean
- Later, this will be very useful for writing generic list operations

Example: List Concatenation

- We can combine two lists into a single list

Example: List Concatenation

- We can combine two lists into a single list
- Polymorphic type

Example: List Concatenation

- We can combine two lists into a single list
- Polymorphic type
    - Works for list with any contents

Example: List Concatenation

- We can combine two lists into a single list
- Polymorphic type
  - Works for list with any contents
  - We never do anything with the contents other than copy

Example: List Concatenation

- We can combine two lists into a single list
- Polymorphic type
  - Works for list with any contents
  - We never do anything with the contents other than copy
  - This function is built into Plait as `append`

Example: List Concatenation

- We can combine two lists into a single list
- Polymorphic type
  - Works for list with any contents
  - We never do anything with the contents other than copy
  - This function is built into Plait as `append`

Example: List Concatenation

- We can combine two lists into a single list
- Polymorphic type
  - Works for list with any contents
  - We never do anything with the contents other than copy
  - This function is built into Plait as `append`

```
(define (concat [xs : (Listof 'elem)]
                [ys : (Listof 'elem)])
        : (Listof 'elem)
  (type-case (Listof 'elem) xs
    [empty
       ys]
    [(cons h t)
       (cons h (concat t ys))]))
```

Example: List Concatenation (ctd.)

Example: List Concatenation (ctd.)

Example: List Concatenation (ctd.)

Example

Example: List Concatenation (ctd.)

Example

```
(concat '(1 2 3) '(4 5 6))
```

Example: List Concatenation (ctd.)

Example

```
(concat '(1 2 3) '(4 5 6))
```

Example: List Concatenation (ctd.)

Example

```
(concat '(1 2 3) '(4 5 6))
(concat '("3" "5") '("0"))
```

```
'(1 2 3 4 5 6)
```

Example: List Concatenation (ctd.)

Example

```
(concat '(1 2 3) '(4 5 6))
(concat '("3" "5") '("0"))
```

```
'(1 2 3 4 5 6)
'("3" "5" "0")
```

Example: List Concatenation (ctd.)

Example

```
(concat '(1 2 3) '(4 5 6))
(concat '("3" "5") '("0"))
(concat '() '(#t))
```

```
'(1 2 3 4 5 6)
'("3" "5" "0")
```

Example: List Concatenation (ctd.)

Example

```
(concat '(1 2 3) '(4 5 6))
(concat '("3" "5") '("0"))
(concat '() '(#t))
```

```
'(1 2 3 4 5 6)
'("3" "5" "0")
'(#t)
```

Example: List Concatenation (ctd.)

Example

```
(concat '(1 2 3) '(4 5 6))
(concat '("3" "5") '("0"))
(concat '() '(#t))
(concat '(#f) '())
```

Results

```
'(1 2 3 4 5 6)
'("3" "5" "0")
'(#t)
```

Example: List Concatenation (ctd.)

Example

```
(concat '(1 2 3) '(4 5 6))
(concat '("3" "5") '("0"))
(concat '() '(#t))
(concat '(#f) '())
```

Results

```
'(1 2 3 4 5 6)
'("3" "5" "0")
'(#t)
'(#f)
```

More Examples

- Demo: Dr. Racket (as time permits)

More Examples

- Demo: Dr. Racket (as time permits)
    - Duplicating each element of a list

More Examples

- Demo: Dr. Racket (as time permits)
  - Duplicating each element of a list
  - “zipping” two lists together

More Examples

- Demo: Dr. Racket (as time permits)
    - Duplicating each element of a list
    - “zipping” two lists together
    - Filtering out odd elements of a list