CMSC 430, Jan 30th 2020

OCaml to Racket

Admin take 2

Admin take 2

- My name: José

Admin take 2

- My name: José
- My email (for now): **`jmct@jmct.cc`**

Admin take 2

- My name: José
- My email (for now): **`jmct@jmct.cc`**
- Website: **`cs.umd.edu/class/spring2020/cmsc430/`**

OCaml, my Caml

OCaml, my Caml

- OCaml is nice.

OCaml, my Caml

- OCaml is nice.
- It’s got all the trimmings of a modern ergonomic programming languages

OCaml, my Caml

- OCaml is nice.
- It’s got all the trimmings of a modern ergonomic programming languages
  - Garbage Collection

OCaml, my Caml

- OCaml is nice.
- It’s got all the trimmings of a modern ergonomic programming languages
  - Garbage Collection
  - Higher-order functions

OCaml, my Caml

- OCaml is nice.
- It’s got all the trimmings of a modern ergonomic programming languages
  - Garbage Collection
  - Higher-order functions
  - Anonymous functions

OCaml, my Caml

- OCaml is nice.
- It's got all the trimmings of a modern ergonomic programming languages
  - Garbage Collection
  - Higher-order functions
  - Anonymous functions
  - Generic types (via parametric polymorphism)

OCaml, my Caml

- OCaml is nice.
- It's got all the trimmings of a modern ergonomic programming languages
  - Garbage Collection
  - Higher-order functions
  - Anonymous functions
  - Generic types (via parametric polymorphism)
  - Pattern matching

OCaml, my Caml

- OCaml is nice.
- It's got all the trimmings of a modern ergonomic programming languages
  - Garbage Collection
  - Higher-order functions
  - Anonymous functions
  - Generic types (via parametric polymorphism)
  - Pattern matching
  - Kind of amazing that it's over 30 years old!

Bop it, twist it, Racket!

Bop it, twist it, Racket!

- In this course we are going to use *Racket*

Bop it, twist it, Racket!

- In this course we are going to use *Racket*
  - Don’t let this worry you, your OCaml skills will apply!

Bop it, twist it, Racket!

- In this course we are going to use *Racket*
  - Don’t let this worry you, your OCaml skills will apply!
  - This lecture and the next will be about learning how to transfer those skills

What a Racket

What a Racket

- In the 90s, the PL group Northeastern University had developed **`PLT Scheme`**, a dialect of LISP

What a Racket

- In the 90s, the PL group Northeastern University had developed **`PLT Scheme`**, a dialect of LISP
- Eventually (in 2010), the differences between **`PLT Scheme`** and **`scheme`** could no longer be reconciled

What a Racket

- In the 90s, the PL group Northeastern University had developed **`PLT Scheme`**, a dialect of LISP
- Eventually (in 2010), the differences between **`PLT Scheme`** and **`scheme`** could no longer be reconciled
- So **`PLT Scheme`** was renamed to **`Racket`**

Why?

Why?

- PLT Scheme was original aimed as a *pedagogical* tool for those learning programming and PLT

Why?

- PLT Scheme was original aimed as a *pedagogical* tool for those learning programming and PLT
- Racket has a notion of ’language levels’

Why?

- PLT Scheme was original aimed as a *pedagogical* tool for those learning programming and PLT
- Racket has a notion of ’language levels’
  - This allows features to be enabled/disabled so that they can be learned/understood individually

Why?

- PLT Scheme was original aimed as a *pedagogical* tool for those learning programming and PLT
- Racket has a notion of 'language levels'
  - This allows features to be enabled/disabled so that they can be learned/understood individually
  - This idea was extended even further to allow user-defined custom languages (which can be used as DSLs!)

Racket Code

Racket code can take a bit to get used to reading,
but its uniform structure makes it easy to learn

Racket Code

Racket code can take a bit to get used to reading, but its uniform structure makes it easy to learn

The code for the first slide looked like this:

```
(slide
   #:title "OCaml to Racket"
   (item "CMSC 430, Jan 30th 2020"))
```

Do people use it?

Do people use it?

- Racket is still used today

Do people use it?

- Racket is still used today
  - Primarily as a research tool (mostly academia, some industry)

Do people use it?

- Racket is still used today
  - Primarily as a research tool (mostly academia, some industry)
  - As a platform for experimenting with all aspects of programming language design

Racket, how to get it:

Racket, how to get it:

- You’ve got some options

Racket, how to get it:

- You've got some options
  - go to **`download.racket-lang.org`**

Racket, how to get it:

- You've got some options
  - go to **`download.racket-lang.org`**
  - Use a package manager (apt/yum/pacman/homebrew/etc.)

Racket, how to get it:

- You've got some options
  - go to **`download.racket-lang.org`**
  - Use a package manager (apt/yum/pacman/homebrew/etc.)
  - Wait until we get a server set up for you all

Racket, how to use it:

Racket, how to use it:

- You’ve got some options!

Racket, how to use it:

- You’ve got some options!
  - Use Dr. Racket, the IDE made and supported by the Racket team

Racket, how to use it:

- You've got some options!
  - Use Dr. Racket, the IDE made and supported by the Racket team
  - Be like me, from the 80's, and develop everything in a text editor

A R.E.P.L. (or repl)

```
430>
```

Arithmetic

Arithmetic

- In OCaml, arithmetic was pretty straightforward:

Arithmetic

- In OCaml, arithmetic was pretty straightforward:

```
> 1 + 2 * 2;;

- : int = 5
```

Arithmetic

- In OCaml, arithmetic was pretty straightforward:

```
> (1) + (2 * 2);;
- : int = 5
```

Arithmetic

- In OCaml, arithmetic was pretty straightforward:

```
> (((1))) + ((2) * 2);;
- : int = 5
```

Arithmetic in Racket

Arithmetic in Racket

- In Racket, an open bracket, **(**, means function application

Arithmetic in Racket

- In Racket, an open bracket, **(**, means function application

```
430>
```

Arithmetic in Racket

- This mean redundant brackets don’t mean what you think!

Arithmetic in Racket

- This mean redundant brackets don't mean what you think!

```
430>
```

Fun(ctions)!

Fun(ctions)!

- Anonymous functions were straightforward in OCaml

Fun(ctions)!

- Anonymous functions were straightforward in OCaml

```
> fun x y -> x + y;;

- : int -> int -> int = <fun>
```

Fun(ctions)!

- Anonymous functions were straightforward in OCaml

```
> (fun x y -> x + y) 3 4;;

- : int = 7
```

Fun(ctions)!

- Anonymous functions were straightforward in OCaml

```
> (fun x y -> x + y) 3;;

- : int -> int = <fun>
```

Fun(ctions)!

- Anonymous functions were straightforward in OCaml

```
> (fun x y -> x + y) 3;;

- : int -> int = <fun>
```

Partial application!

Fun in Racket

Fun in Racket

- In OCaml we had: **`fun x y -> x + y`**

Fun in Racket

- In OCaml we had: **`fun x y -> x + y`**
- What's that look like in Racket?

Fun in Racket

- In OCaml we had: **`fun x y -> x + y`**
- What's that look like in Racket?

```
430>
```

Get the clickers out

Get the clickers out

- What's this mean, in Racket?

```
430> (λ (x)
        (λ (y)
           (+ x y))) 3 4
```

Get the clickers out

- What's this mean, in Racket?

```
430> (λ (x)
        (λ (y)
           (+ x y))) 3 4
```

  - A) 7

Get the clickers out

- What's this mean, in Racket?

```
430> (λ (x)
        (λ (y)
           (+ x y))) 3 4
```

  - A) 7
  - B) error

Get the clickers out

- What's this mean, in Racket?

```
430> (λ (x)
        (λ (y)
           (+ x y))) 3 4
```

  - A) 7
  - B) error
  - C) Something else

The right way

The right way

```
430> ((λ (x)
        (λ (y)
           (+ x y))) 3 4)
```

Fun in Racket

Fun in Racket

- In OCaml we had:

```
(fun (x, y) -> x + y) (3, 4)
```

Fun in Racket

- In OCaml we had:
  ```
  (fun (x, y) -> x + y) (3, 4)
  ```
- What's that look like in Racket?

Fun in Racket

- In OCaml we had:

```
(fun (x, y) -> x + y) (3, 4)
```

- What's that look like in Racket?

```
430> ((λ (x y)
        (+ x y)) ??)
```

Let’s take a look

Let’s take a look

- Definitions in OCaml used **`let`**

Let’s take a look

- Definitions in OCaml used **`let`**

```
> let x = 3;;
val x : int = 3
```

Let’s take a look

- Definitions in OCaml used **`let`**

```
> let y = 4;;
val y : int = 4
```

Let’s take a look

- Definitions in OCaml used **`let`**

```
> x + y;;
- : int = 7
```

Let’s take a look

- Definitions in OCaml used **`let`**
- This is true for functions, too

Let’s take a look

- Definitions in OCaml used **`let`**
- This is true for functions, too

```
> let mul a b = a * b;;
val mul : int -> int -> int = <fun>
```

Let’s take a look

- Definitions in OCaml used **`let`**
- This is true for functions, too

```
> let mul a b = a * b;;
val mul : int -> int -> int = <fun>
> mul x y;;
- : int = 12
```

Defs in Racket

Defs in Racket

- In Racket we define things with **`define`**

Defs in Racket

- In Racket we define things with **`define`**

```
430> (define x 3)
     (define y 4)
     (+ x y)
```

Defs in Racket

- In Racket we define things with **`define`**
- Also true for functions

Defs in Racket

- In Racket we define things with **`define`**
- Also true for functions

```
430> (define mul
       (λ (a b)
          (* a b)))
     (mul 3 4)
```

Defs in Racket

- There's a shorthand for function definitions that lets us avoid the lambda

```
(define (mul a b)
  (* a b))
```

Lists

Lists

- Lists are the bread-and-butter of functional programming

Lists

- Lists are the bread-and-butter of functional programming

```
> 1 :: 2 :: 3 :: [];;

- : int list = [1; 2; 3]
```

Pros and Cons

Pros and Cons

- What's that look like in Racket?

Pros and Cons

- What's that look like in Racket?

```
430> (cons 1 (cons 2 (cons 3 '())))
```

Pros and Cons

- Luckily there's a helper function for this

Pros and Cons

- Luckily there's a helper function for this

```
430> (list 1 2 3)
```

Get the clickers out

Get the clickers out

- Is this a valid OCaml definition?
- **`let xs = ["jazz"; 1959];;`**

Get the clickers out

- Is this a valid OCaml definition?
- **`let xs = ["jazz"; 1959];;`**
  - A) Yes

Get the clickers out

- Is this a valid OCaml definition?
- **`let xs = ["jazz"; 1959];;`**
  - A) Yes
  - B) No

Get the clickers out

- Is this a valid OCaml definition?
- **`let xs = ["jazz"; 1959];;`**
  - A) Yes
  - B) No
  - C) I don't understand the question and I won't respond to it.

Pros of Cons

- Racket is Dynamically typed, so the following is perfectly valid

Pros of Cons

- Racket is Dynamically typed, so the following is perfectly valid

```
430> (list "jazz" 1959)
```

Pairs _are_ Cons

- Because Racket is dynamically typed, constructing pairs is the same thing as constructing lists

Pairs _are_ Cons

- Because Racket is dynamically typed, constructing pairs is the same thing as constructing lists

```
430> (cons "jazz" 1959)
       (cons "hip hop" 2015)
```

Assignment #1

Assignment #1

- Learning about a Programming Language

Assignment #1

- Learning about a Programming Language
- Email me the solution, ensuring that the subject starts with **`[Assignment 1]`**

Assignment #1

- Learning about a Programming Language
- Email me the solution, ensuring that the subject starts with **`[Assignment 1]`**
- Details are posted on the website (including which languages you can't discuss)

Assignment #1

- Learning about a Programming Language
- Email me the solution, ensuring that the subject starts with **`[Assignment 1]`**
- Details are posted on the website (including which languages you can't discuss)
- The first few slides of this lecture (about Racket) is basically the level of detail I'm looking for

Assignment #1

- Learning about a Programming Language
- Email me the solution, ensuring that the subject starts with **`[Assignment 1]`**
- Details are posted on the website (including which languages you can't discuss)
- The first few slides of this lecture (about Racket) is basically the level of detail I'm looking for
- Go, you're free.

OCaml to Racket, Part 2

Lists (cons) of pairs (cons)

Lists (cons) of pairs (cons)

- Structured data is nice, let’s make a dictionary.

Lists (cons) of pairs (cons)

- Structured data is nice, let's make a dictionary.

```
430> (require "genre-years.rkt")
```

Destructors

```
430> (require "genre-years.rkt")
```

Destructors 2

Destructors 2

- What would **`car`** and **`cdr`** do on lists?

Destructors 2

- What would **`car`** and **`cdr`** do on lists?
  - **`(car '(1 2 3)) ==> ????`**
  - **`(cdr '(1 2 3)) ==> ????`**

Destructors 3

Destructors 3

- Do yourself a favor

Destructors 3

- Do yourself a favor

```
(define fst car)
(define snd cdr)
```

Pattern Matching!

- Just like in OCaml, we can pattern match to help us define functions

Pattern Matching!

- Just like in OCaml, we can pattern match to help us define functions

```
(define (swap p)
  (match p
    [(cons x y) (cons y x)]))
```

Pattern Matching!

- Just like in OCaml, we can pattern match to help us define functions

```
(define (is-two-or-four n)
  (match n
    [2 #t]
    [4 #t]
    [_ #f]))
```

Pattern Matching!

- Just like in OCaml, we can pattern match to help us define functions

```
(define (sum xs)
  (match xs
    ['() 0]
    [(cons y ys)
     (+ x (sum xs))]))
```

Datatypes

Datatypes

- One of the more elegant features of typed-functional PLs is algebraic datatypes

Datatypes

- One of the more elegant features of typed-functional PLs is algebraic datatypes
  - **`type bt = Leaf | Node of int * bt * bt`**

Datatypes

- One of the more elegant features of typed-functional PLs is algebraic datatypes
  - **`type bt = Leaf | Node of int * bt * bt`**
    - Defining and then pattern matching on ADTs is a very powerful tool for reasoning about programs

Datatypes

- One of the more elegant features of typed-functional PLs is algebraic datatypes
- Racket does not have ADTs directly, but we can get close with **`struct`**

Datatypes

- One of the more elegant features of typed-functional PLs is algebraic datatypes
- Racket does not have ADTs directly, but we can get close with **`struct`**
  - **`struct`** lets us define a structured value

Datatypes

- One of the more elegant features of typed-functional PLs is algebraic datatypes
- Racket does not have ADTs directly, but we can get close with **`struct`**
  - **`struct`** lets us define a structured value
  - i.e. like a single constructor from a datatype in OCaml

Datatypes

- One of the more elegant features of typed-functional PLs is algebraic datatypes
- Racket does not have ADTs directly, but we can get close with **`struct`**
  - **`struct`** lets us define a structured value
  - i.e. like a single constructor from a datatype in OCaml
  - But then we can use it for pattern matching!

Structs

Structs

- Let’s try to emulate the binary tree we showed in OCaml

Structs

- Let's try to emulate the binary tree we showed in OCaml

```
(struct leaf ())
```

Structs

- Let's try to emulate the binary tree we showed in OCaml

```
(struct leaf ())

(struct node (i left right))
```

Structs in the REPL

```
430> (struct leaf ())
     (struct node (i left right))
```

Pattern matching on structs

Pattern matching on structs

- Defining a function that checks whether a tree is empty

Pattern matching on structs

- Defining a function that checks whether a tree is empty

```
(define (bt-empty? bt)
  (match bt
    [(leaf)         #t]
    [(node _ _ _) #f]))
```

Defining accessors

Defining accessors

```
(define (get-elem bt)
  (match bt
    [(leaf)          '()]
    [(node i _ _) (cons i '())]))
```

It may tick of you off, but symbols matter

It may tick of you off, but symbols matter

- Symbols are preceded by the '

It may tick of you off, but symbols matter

- Symbols are preceded by the '
- You don't have to define them beforehand, you can just use them:

It may tick of you off, but symbols matter

- Symbols are preceded by the `'`
- You don’t have to define them beforehand, you can just use them:
  - `'All 'of 'these 'are 'symbols`

It may tick of you off, but symbols matter

- Symbols are preceded by the `'`
- You don’t have to define them beforehand, you can just use them:
  - `'All 'of 'these 'are 'symbols`
- Equality on symbols is what you might expect:

It may tick of you off, but symbols matter

- Symbols are preceded by the `'`
- You don’t have to define them beforehand, you can just use them:
  - `'All 'of 'these 'are 'symbols`
- Equality on symbols is what you might expect:

```
430> (equal? 'Λ 'Λ)
     (equal? 'José 'Jose)
```

A Symbol unlike any other

A Symbol unlike any other

- In compilers we often need symbols that can’t clash with any existing symbols

A Symbol unlike any other

- In compilers we often need symbols that can’t clash with any existing symbols
  - Anything that gives you such a symbol is considered a source of ’fresh names’

A Symbol unlike any other

- In compilers we often need symbols that can't clash with any existing symbols
  - Anything that gives you such a symbol is considered a source of 'fresh names'
- In Racket:

```
430> (gensym)
     (gensym)
     (gensym)
```

For the enumerated type in your life

For the enumerated type in your life

- If OCaml we could write the following type:

For the enumerated type in your life

- If OCaml we could write the following type:

```
type Beatles = JohnL   | PaulM
             | GeorgeH | RingoS
             | BillyP  | GeorgeM
```

For the enumerated type in your life

- If OCaml we could write the following type:

```
type Beatles = JohnL   | PaulM
             | GeorgeH | RingoS
             | BillyP  | GeorgeM
```

- In Racket:

```
(define beatles (list 'JohnL 'PaulM
                      'GeorgeH 'RingoS
                      'BillyP 'GeorgeM))
(define (beatle? p)
  (member p beatles))
```

Code = Data

Code = Data

- We've already seen one of Racket's most powerful features: Quote/Unquote

Code = Data

- We've already seen one of Racket's most powerful features: Quote/Unquote
  - Now we're going to look at it a little closer

Code = Data

- We've already seen one of Racket's most powerful features: Quote/Unquote
  - Now we're going to look at it a little closer

```
'(x y z) == (list 'x 'y 'z)
```

Code = Data

- We've already seen one of Racket's most powerful features: Quote/Unquote
  - Now we're going to look at it a little closer

```
'(x y z) == (list 'x 'y 'z)
```

- In Racket `'` is known as **`quote`**

Code = Data

Code = Data

- A quoted thing can always be represented as an unquoted thing by pushing the ' `inwards'

Code = Data

- A quoted thing can always be represented as an unquoted thing by pushing the **'** `inwards'
- **'** `stop' at symbols (i.e. **'PaulM**) or empty brackets **'()**

Code = Data

- A quoted thing can always be represented as an unquoted thing by pushing the **'** `inwards'
- **'** `stop' at symbols (i.e. **'PaulM**) or empty brackets **'()**
- **'** goes away at booleans, strings, and numbers. So:

Code = Data

- A quoted thing can always be represented as an unquoted thing by pushing the **'** `inwards'
- **'** `stop' at symbols (i.e. **'PaulM**) or empty brackets **'()**
- **'** goes away at booleans, strings, and numbers. So:
  - **'3 == 3**
  - **'"String" == "String"**
  - **'#t == #t**

Oh, pairs.

Oh, pairs.

- If **`'(1 2)`** means **`(list '1 '2)`**

Oh, pairs.

- If **`'(1 2)`** means **`(list '1 '2)`**
  - How would we write something that means **`(cons '1 '2)`**?

Oh, pairs.

- If **`'(1 2)`** means **`(list '1 '2)`**
  - How would we write something that means **`(cons '1 '2)`**?
  - ... We have to add syntax :(

Oh, pairs.

- If **`'(1 2)`** means **`(list '1 '2)`**
  - How would we write something that means **`(cons '1 '2)`**?
  - ... We have to add syntax :(
- **`'(1 . 2)`**

When you what to quote, but only kinda.

When you what to quote, but only kinda.

- If you use ` it works a lot like '

When you what to quote, but only kinda.

- If you use ` it works a lot like '
  - **`(a b c) == (list `a `b `c)**

When you what to quote, but only kinda.

- If you use ` it works a lot like '
  - **`(a b c) == (list `a `b `c)**
- In fact, there is only one difference

When you what to quote, but only kinda.

- If you use `` ` `` it works a lot like `'`
  - **`` `(a b c) == (list `a `b `c) ``**
- In fact, there is only one difference
  - `` ` `` works exactly like quote, unless it encounters a `,`

When you what to quote, but only kinda.

- If you use `` ` `` it works a lot like `'`
  - **`` `(a b c) == (list `a `b `c) ``**
- In fact, there is only one difference
  - `` ` `` works exactly like quote, unless it encounters a `,`
  - **`` `,e == e ``**

When you what to quote, but only kinda.

- If you use `` ` `` it works a lot like `'`
  - **`` `(a b c) == (list `a `b `c) ``**
- In fact, there is only one difference
  - `` ` `` works exactly like quote, unless it encounters a `,`
  - **`` `,e == e ``**
- These are known as **`quasiquote`** and **`unquote`**, respectively.

- What result should this give us?

```
430> `(+ 1 ,(+ 1 1))
```

- What about this?

```
430> `(+ 1 ,(+ 1 1) 1)
```

Flipping the bit on binary trees

Flipping the bit on binary trees

- We showed how to do binary trees with structs

Flipping the bit on binary trees

- We showed how to do binary trees with structs
- Another pattern in Racket is to encode ADTs as s-expressions (all the things you can quote/unquote)

Flipping the bit on binary trees

- We showed how to do binary trees with structs
- Another pattern in Racket is to encode ADTs as s-expressions (all the things you can quote/unquote)

```
430> 'leaf
'(node 3 leaf leaf)
```

Flipping the bit on binary trees

- We showed how to do binary trees with structs
- Another pattern in Racket is to encode ADTs as s-expressions (all the things you can quote/unquote)

```
430> 'leaf
'(node 3 leaf leaf)
```

- Note that **`leaf`** and **`node`** are just symbols!

Let’s study this code together

Let's study this code together

```
(define (bt-height bt)
  (match bt
    [`leaf 0]
    [`(node ,_ ,left ,right)
     (+ 1 (max (bt-height left)
               (bt-height right)))]))
```

To catch them is my real test.

```
430> (require rackunit)
(check-equal? (* 2 3) 7)
```

Some final thoughts

Some final thoughts

- Read the lecture notes!

Some final thoughts

- Read the lecture notes!
  - There is material on testing racket code, and how to define and import modules