Pattern Matching

Yi Dai

June 8, 2012

Motivating Example 1 --- Debitalization (1/2)

bit	$\xrightarrow{debitalize}$	truth
0		false
1		true

Debitalization (1a/2)

Racketize it

Debitalization (1a/2)

Racketize it

Data

- number: `0,1`
- boolean: `false,true`

Debitalization (1a/2)

Racketize it

Data

- number: 0, 1
- boolean: false, true

Code

```
;; debit : number -> boolean
;; debitalizes a bit, via equality test
(define (debit b)
  (cond ((= b 0) false)
        ((= b 1) true ) ) )
```

Debitalization (1b/2)

Bit operators

bit operator	boolean operator
NOT	$\neg$
AND	$\wedge$
OR	$\vee$

Debitalization (1b/2)

Bit operators

bit operator	boolean operator
NOT	$\neg$
AND	$\wedge$
OR	$\vee$

Racket code

```
;; NOT : number -> number
;; negates a bit, via equality test
;; ...
```

Debitalization (1b/2)

Bit operators

bit operator	boolean operator
NOT	$\neg$
AND	$\wedge$
OR	$\vee$

Racket code

```
;; NOT : number -> number
;; negates a bit, via equality test
;; ...
```

AND and OR ...

Debitalization (1c/2)

...

Debitalization (1c/2)

...

```
;; AND : number number -> number
;; conjoins two bits, via equality test
(define (AND b1 b2)
  (cond ((and (= b1 0) (= b2 0)) 0)
        ((and (= b1 0) (= b2 1)) 0)
        ((and (= b1 1) (= b2 0)) 0)
        ((and (= b1 1) (= b2 1)) 1) ) )

;; OR : number number -> number
;; disjoins two bits, via equality test
;; ...

(debit (NOT (OR (AND 0 1) (AND 1 0))))
```

A First Taste of Pattern Matching

Refactor it

```
;; debit : number -> boolean
;; debitalizes a bit, via pattern matching
(define (debit b)
  (match b
    (0 false)
    (1 true ) ) )
```

A First Taste of Pattern Matching

Refactor it

```
;; debit : number -> boolean
;; debitalizes a bit, via pattern matching
(define (debit b)
  (match b
    (0 false)
    (1 true ) ) )

;; NOT : number -> number
;; negates a bit, via pattern matching
;; ...
```

A First Taste of Pattern Matching

Refactor it

```
;; debit : number -> boolean
;; debitalizes a bit, via pattern matching
(define (debit b)
  (match b
    (0 false)
    (1 true ) ) )

;; NOT : number -> number
;; negates a bit, via pattern matching
;; ...
```

AND and OR ...

Nested Pattern Matching

...

```
;; AND : number number -> number
;; conjoins two bits, via pattern matching
(define (AND b1 b2)
  (match b1
    (0 0)
    (1 (match b2
         (0 0)
         (1 1) ) ) ) )

;; OR : number number -> number
;; disjoins two bits, via pattern matching
;; ...

(debit (NOT (OR (AND 0 1) (AND 1 0))))
```

Matching Literals

Racket literals

- booleans
- numbers

Matching Literals

Racket literals

- booleans
- numbers
- characters

Matching Literals

Racket literals

- booleans
- numbers
- characters
- strings
- ...

Matching Literals

Racket literals

- booleans
- numbers
- characters
- strings
- ...

matching literals $=$ equality test

Tastes good?

Tastes good?

Clean code

Example 1 --- Debitalization (2/2)

bit	$\xrightarrow{debitalize}$	truth
0		false
1		true

Example 1 --- Debitalization (2/2)

bit	$\xrightarrow{debitalize}$	truth
0		false
1		true

bit stream	$\xrightarrow{debitalize}$	truth stream
0 1 ...		false true ...

Debitalization (2a/2)

Racketize it

Debitalization (2a/2)

Racketize it

Data

- list of numbers: `(list 0 1 ...)`
- list of booleans: `(list false true ...)`

Debitalization (2a/2)

Racketize it

Data

- list of numbers: `(list 0 1 ...)`
- list of booleans: `(list false true ...)`

Code

```
;; debits : (listof number) -> (listof boolean)
;; debitalizes a bit stream, via isomorphism test
(define (debits bs)
  (cond ((empty? bs) bs)
        (else (let ((b (first bs))
                    (bs (rest bs)) )
                (cons (debit b)
                      (debits bs) ) ) ) ) )
```

Debitalization (2b/2)

Bit-wise NOT, AND and OR

Debitalization (2b/2)

Bit-wise NOT, AND and OR

Racket code

```
;; NOTs : (listof number) -> (listof number)
;; bit-wise negates a bit stream, via isomorphism test
;; ...
```

Debitalization (2b/2)

Bit-wise NOT, AND and OR

Racket code

```
;; NOTs : (listof number) -> (listof number)
;; bit-wise negates a bit stream, via isomorphism test
;; ...
```

`ANDs` and `ORs` ...

Debitalization (2c/2)

…

Debitalization (2c/2)

...

```
;; ANDs : (listof number) (listof number) -> (listof number)
;; bit-wise conjoins two bit streams, via isomorphism test
(define (ANDs bs1 bs2)
  (cond ((or (empty? bs1) (empty? bs2)) empty)
        (else (let ((b1 (first bs1))
                    (bs1 (rest bs1))
                    (b2 (first bs2))
                    (bs2 (rest bs2)) )
                (cons (AND b1 b2)
                      (ANDs bs1 bs2) ) ) ) ) )

;; ORs : (listof number) (listof number) -> (listof number)
;; bit-wise disjoins two bit streams, via isomorphism test
;; ...

(debits (NOTs (ORs (ANDs (list 0 1 0))
                   (ANDs (list 1 0 1 0)) ) ) )
```

A Second Taste of Pattern Matching

Refactor it

```
;; debits : (listof number) -> (listof boolean)
;; debitalizes a bit stream, via pattern matching
(define (debits bs)
  (match bs
    ((list) bs)
    ((list b bs ...) (cons (debit b)
                           (debits bs) ) ) ) )
```

A Second Taste of Pattern Matching

Refactor it

```
;; debits : (listof number) -> (listof boolean)
;; debitalizes a bit stream, via pattern matching
(define (debits bs)
  (match bs
    ((list) bs)
    ((list b bs ...) (cons (debit b)
                           (debits bs) ) ) ) ) )

;; NOTs : number -> number
;; bit-wise negates a bit stream, via pattern matching
;; ...
```

A Second Taste of Pattern Matching

Refactor it

```
;; debits : (listof number) -> (listof boolean)
;; debitalizes a bit stream, via pattern matching
(define (debits bs)
  (match bs
    ((list) bs)
    ((list b bs ...) (cons (debit b)
                           (debits bs) ) ) ) )

;; NOTs : number -> number
;; bit-wise negates a bit stream, via pattern matching
;; ...
```

AND and OR ...

Nested Pattern Matching

...

```
;; ANDs : (listof number) (listof mumber) -> (listof number)
;; bit-wise conjoins two bit streams, via pattern matching
(define (ANDs bs1 bs2)
  (match bs1
    ((list) bs1)
    ((list b1 bs1 ...)
     (match bs2
       ((list) bs2)
       ((list b2 bs2 ...) (cons (AND b1 b2)
                                (ANDs bs1 bs2) ) ) ) ) ) )

;; OR : (listof number) (listof number) -> (listof number)
;; bit-wise disjoins two bit streams, via pattern matching
;; ...

(debits (NOTs (ORs (ANDs (list 0 1 0))
                   (ANDs (list 1 0 1 0)) ) ) )
```

Matching Built-in Data Structures

Racket built-in data structures

- lists

Matching Built-in Data Structures

Racket built-in data structures

- lists
- pairs

Matching Built-in Data Structures

Racket built-in data structures

- lists
- pairs
- vectors
- ...

Matching Built-in Data Structures

Racket built-in data structures

- lists
- pairs
- vectors
- ...

matching (built-in) data structures = isomorphism test

Tastes good?

Tastes good?

Clear code

Motivating Example 2 --- Poker

card	
rank	A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K
suit	♠, ♡, ♢, ♣, ★

Poker (1a)

Racketize it

Poker (1a)

Racketize it

Data

- structure: `struct`

Poker (1a)

Racketize it

Data

- structure: `struct`
- string: `"A"`, `"2"`, `"♠"`, ...

Poker (1a)

Racketize it

Data

- structure: `struct`
- string: `"A"`, `"2"`, `"♠"`, ...

Code

```
(define-struct card (rank suit))
```

Poker (1b)

```
;; rank-name : string -> string
;; echoes the name of a rank
;; ...

;; suit-name : string -> string
;; echoes the name of a suit
;; ...

;; card-name : card -> string
;; echoes the name of a card
(define (card-name c)
  (let ((r (card-rank c))
        (s (card-suit c)) )
    (string-append (rank-name r) " of "
                   (suit-name s) "s" ) ) )
```

A Third Taste of Pattern Matching (1c)

Refactor it

```
;; card-name : card -> string
;; echoes the name of a card
(define (card-name c)
  (match c
    ((struct card (r s))
     (string-append (rank-name r) " of "
                    (suit-name s) "s") ) ) )
```

Matching User-defined Data Structures

Racket user-defined data structures

- via `struct`

Matching User-defined Data Structures

Racket user-defined data structures

- via `struct`

matching (user-defined) data structures $=$ isomorphism test

Tastes good?

Tastes good?

Concise code

An Explicit Way to Apply a Function

The usual way

- `(+ 1 2 3)`
- `(length (list 1 2 3))`

An Explicit Way to Apply a Function

The usual way

- `(+ 1 2 3)`
- `(length (list 1 2 3))`

Say `apply` explicitly

- `(apply + (list 1 2 3))`

An Explicit Way to Apply a Function

The usual way

- `(+ 1 2 3)`
- `(length (list 1 2 3))`

Say `apply` explicitly

- `(apply + (list 1 2 3))`
- `(apply length (list (list 1 2 3)))`

Use it only when necessary ...

The Wonder of Variadic Functions

```
(+) = (apply + (list))
```

The Wonder of Variadic Functions

```
(+) = (apply + (list))
(+ 1) = (apply + (list 1))
```

The Wonder of Variadic Functions

```
(+) = (apply + (list))
(+ 1) = (apply + (list 1))
(+ 1 2) = (apply + (list 1 2))
```

The Wonder of Variadic Functions

```
(+) = (apply + (list))
(+ 1) = (apply + (list 1))
(+ 1 2) = (apply + (list 1 2))
(+ 1 2 3) = (apply + (list 1 2 3))
...
```

The Wonder of Variadic Functions

```
(+) = (apply + (list))
(+ 1) = (apply + (list 1))
(+ 1 2) = (apply + (list 1 2))
(+ 1 2 3) = (apply + (list 1 2 3))
```

...

Argument list

The Magic behind Variadic Functions

```
;; sum : number ... -> number
;; sums an arbitrary number of numbers
(define (sum . ns)
  (cond ((empty? ns) 0)
        (else (+ (first ns)
                 (apply sum (rest ns)) ) ) ) )
```

The Magic behind Variadic Functions

```
;; sum : number ... -> number
;; sums an arbitrary number of numbers
(define (sum . ns)
  (cond ((empty? ns) 0)
        (else (+ (first ns)
                 (apply sum (rest ns)) ) ) ) )
```

Refactor it?

Functions with Optional Arguments

```
;; greet-person : string string ... -> string
(define (greet-person g . ps)
  (match ps
    ((list) (string-append g "!"))
    ((list p) (string-append g ", " p "!"))
    ((list p ps ...)
     (string-append g ", " p "! "
                    (apply greet-person (cons g ps))
```

Functions with Optional Arguments

```
;; greet-person : string string ... -> string
(define (greet-person g . ps)
  (match ps
    ((list) (string-append g "!"))
    ((list p) (string-append g ", " p "!"))
    ((list p ps ...)
     (string-append g ", " p "! "
                    (apply greet-person (cons g ps))))))
```

An *optional* feature