Pattern matching. Macros

May 27, 2014

Content of this lecture

- Pattern matching
  - modern feature in functional programming, that allows to extract the components of a compound value
  - pattern matching in RACKET: the `match` form
  - Quasiquoting: `quasiquote`, `unquote`, and `unquote-splicing`
- Macros = an extension mechanism of the language compiler
  - pattern-based macros
  - identifier macros

Values

Value = expression that evaluates to itself

- Values do not contain function calls
- Data type = set of values with common properties.
  - ▷ predefined data types: integers, booleans, pairs, lists, vectors, hash maps ...
  - ▷ data types defined by the user with the `struct` form (also known as structures)
- Every value *v* is either
  - A simple value, e.g., `1`, `3/4`, `#t`, etc.
  - A compound value, uniquely described by the way it was constructed. E.g.,
    - `(list 1 2)` (or `'(1 2)`): list of `1` and `2`
    - `(pair 1 2)` (or `'(1 . 2)`): pair of `1` and `2`
    - `(vector 1 2)` (or `'#(1 2)`): vector of `1` and `2`
    - `(pos 7 8)`: instance of a user-defined structure `pos`

In general, the value of a compound data type is of the form `(`*constr* v_1 ... v_n`)` where *constr* is a constructor and $v_1, \ldots, v_n$ are its component values.

Multiple values

A RACKET expression can evaluate to multiple values, in the same way that a function can accept multiple arguments.

- `(values` $expr_1$ ... $expr_n$`)`
  returns the values of expressions $expr_1, \ldots, expr_n$
- `(multiple-values` id_1 ... id_n $expr$`)`
  evaluates *expr* which is expected to produce *n* multiple values $v_1, \ldots, v_n$ which are assigned to $id_1, \ldots, id_n$ respectively.

Example

```
> (values 1 (+ 2 3))
1
5
```

```
> (define-values (x y z)
      (values 1 2 3))
> x
1
> (list y z)
'(2 3)
```

Multiple values
A more complex example

Define two functions `get-clock` and `put-clock!` that share a private variable `private-clock` initialised with `0`, such that:

- `(get-clock)` returns the current value of `private-clock`
- `(put-clock!` *v*`)` sets the value of `private-clock` to the value of *v*.

```
> (define-values (get-clock put-clock!)
    (let ([private-clock 0])
      (values (lambda () private-clock)
              (lambda (v) (set! private-clock v)))))
>  (put-clock! 7)
> (get-clock)
7
```

Extracting values from compound values

- **Method 1**: with recognisers and selectors.
- **Method 2**: by pattern matching.

Illustrative example

Define a function `(get-first` *v*`)` that takes as input a value *v*, and

- returns the first component of *v*, if *v* is a pair of non-empty list or vector.
- returns the string "empty list" if *v* is empty list
- returns the string "empty vector" if *v* is empty vector
- returns the string "don't know" in all other cases.

Extracting values from compound values

Method 1: with recognisers and selectors

```
(define (get-first v)
    (if (pair? v)
        (car v)
        (if (null? v)
            "empty list"
            (if (list? v)
                (car v)
                (if (vector? v)
                    (if  (> (vector-length v) 0)
                        (vector-ref v 0)
                        "empty vector")
                    "don't know")))))
```

1. Ugly code: lots of nested `if`s
2. Any attempt to apply a selector to input of the wrong kind fails with an error.

Extracting values from compound values
Method 2: by pattern matching

```
(define (get-first v)
  (match v
    [(cons v1 _) v1]  ; matches any list or pair with a first value v1
    [(vector v1 _ ...) v1]  ; matches any vector with a first value v1
    [(list) "empty list"]   ; matches empty list
    [(vector) "empty vector"]  ; matches empty vector
    [_ "don't know"]))         ; matches anything else
```

We can collapse the first two matching clauses into one, using `or` between patterns:

```
(define (get-first v)
  (match v
    [(or (cons v1 _) (vector v1 _ ...)) v1]
    [(list) "empty list"]
    [(vector) "empty vector"]
    [_ "don't know"]))
```

The red-colored expressions are called patterns:

- a non-constructor identifier `x` is called *pattern variable*: it matches any value, and the matched value is bound to `x`
- _ is a *catch-all* pattern: it matches any value without performing any variable binding
- *p* . . . is used to match sequences of consecutive values in a list or vector.
  - If *p* contains pattern variables, they are bound as many times as *p* matches consecutive values
  - The value of a pattern variable of in *p* . . . is the list of bindings it received by matching *p* with consecutive values.
- (*constr* p_1 ... p_n) matches any value built with constructor *constr* from values $v_1, \ldots, v_n$ that match $p_1, \ldots, p_n$, respectively.

```
(match expr
    [pattern_1  body_1]
    ...
    [pattern_n  body_n])
```

takes the value v of $expr$ and tries to match each of the patterns $pattern_1, \ldots, pattern_n$ with v, in this order.

1. If $pattern_i$ is the first pattern that matches v then
   - The corresponding body $body_i$ is evaluated in an environment which binds the pattern variables to their corresponding values.
   - The computed value is returned as result of the `match` form.
2. If no $pattern_i$ matches v, an error message is issued.

Pattern matching examples

```
> (match (vector 1 2)
    [(list x _ ...) x]
    [(vector x _ ...) x])
1
> (struct shoe (size colour))  ; user-defined data type
> (struct hat (size style))    ; user-defined data type
> (match (shoe 42 'brown)  ; return the size of hat or shoe
     [(hat x _) x]
     [(shoe x _) x])
>  (match '(1 2 3 4) [(list 1 x ... _) x])
'(2 3)
```

Note that in the last example, the pattern `(list 1 x ... _)` matches the list `'(1 2 3 4)` as follows:

- It matches 1 with 1
- It matches `x ...` with the sequence of values `2 3` and binds `x` to the list `'(2 3)`
- It matches _ with 4.

More pattern matching examples

```
> ; x appears in x ... ⇒ it is bound to the list of all its bindings
   (match '((1 2 3) (4 5 6))
     [(list (list _ x ...) ...) x])
'((2 3) (5 6))
> ;  x does not appear inside a sequence pattern
   ; therefore, all the values it matches must be the same
> (match '(1 1) [(list x x) x] [_ "different"])
1
> (match '(1 2) [(list x x) x] [_ "different"])
"different"
```

The printed forms of lists and vectors use the quote mark ' to make them easier to read:

```
> (define v                          ; bind v to a vector
     (vector 1 (list 2 3) (vector 'a 'b)))
> v
'#(1 (2 3) #(a b))
> (define l                          ; bind l to a list
     (list 'alpha v))
> l
'(alpha #(1 (2 3) #(a b)))
> (list l v)
'((alpha #(1 (2 3) #(a b))) #(1 (2 3) #(a b)))
```

- The printed form of a list is of the form `'(...)`
- The printed form of a vector is of the form `'#(...)`

Quasiquoting

MAIN IDEA: We wish to use a simplified syntax (like quoted expressions) to write patterns and expressions.

The quasiquoting mechanism for expressions

`` `datum `` is the same as the printed form `'datum`, except for the fact that:

- Any subexpression `,expr` of `datum` is replaced by the value of `expr`.
  - ▷ `,expr` abbreviates `(unquote expr)`
- Any subexpression `,@expr` of `datum` is replaced by the (possibly empty) sequence of values v_1 ... v_n if `expr` evaluates to the list of values `(list` v_1 ... v_n`)`.
  Such a replacement is called *splicing*.
  - ▷ `,@expr` abbreviates `(unquote-splicing expr)`

`` `datum `` abbreviates `(quasiquote datum)`
`'datum` abbreviates `(quote datum)`

Quasiquoting of expressions
Examples

```
> (define v '(1 . a))
> (define l '(a 2 (b 3)))
> `(,v ,l)
'((1 . a) (a 2 (b 3)))
> `(v l)
'(v l)
> `(,v ,@l)
'((1 . a) a 2 (b 3))
> `(1 2 ,@(list (+ 1 2) (- 5 1)))
'(1 2 3 4)
```

Remarks

1) `quote` (`'`) freezes completely evaluation of its subexpressions.
2) `quasiquote` (`` ` ``) freezes evaluation of its subexpressions, except those unquoted with `unquote` (`,`) and `unquote-splicing` (`,@`)

Quasiquoting of expressions

Example

A recursive definition with `quasiquote` and `unquote-splicing` of the list `'(1 2 ...` *n*`)`:

```
 > (define (upto n)
        (if (zero? n)
            '(0)
            `(,@(upto (- n 1)) ,n)))
> (upto 10)
'(0 1 2 3 4 5 6 7 8 9 10)
```

If a `quasiquote` form appears within an enclosing `quasiquote` form, then the inner `quasiquote` effectively cancels one layer of `unquote` and `unquote-splicing` forms, so that a second `unquote` or `unquote-splicing` is needed.

```
> `(1 2 `(,(+ 1 2)))
'(1 2 `(,(+ 1 2)))
> `(1 2 `(,,(+ 1 2)))
'(1 2 `(,3))
> `(1 2 `(,(+ 1 2) ,,(- 5 1)))
'(1 2 `(,(+ 1 2) ,4))
```

Pattern matching with `quasiquote` expressions

- `` `datum `` can be used as a pattern, if the following convention is respected: The pattern variables in `datum` are `unquote`d

Example

```
> (define (get-first v)
    (match v
      [(or `(,v1 . ,_) `#(,v1 ,_ ...)) v1]
      ['() "empty list"]
      ['#() "empty vector"]
      [_ "don't know"]))
> (get-first '#())
"empty vector"
> (get-first '((a #t) . c))
'(a #t)
> (get-first 1)
"don't know"
```

Macro = syntactic form with an associated *transformer* that *expands* it into an expression without macros.

- In RACKET, all syntactic forms are transformed in 2 phases:
  1. Expansion: All macros are expanded by their corresponding transformers ⇒ syntactic form without macros
  2. Evaluation: The expanded syntactic form is reduced to a value by the interpreter of RACKET

$$\underbrace{\begin{array}{c}\text{syntactic form}\\ \text{with macros}\end{array} \longrightarrow \boxed{\begin{array}{c}\text{macro}\\ \text{transformers}\end{array}} \longrightarrow \begin{array}{c}\text{expanded form}\\ \text{without macros}\end{array}}_{\text{expansion}} \underbrace{\longrightarrow \boxed{\text{interpreter}} \longrightarrow \text{value}}_{\text{evaluation}}$$

- Every macro has a *macro id*
- A macro definition indicates the transformer associated with the syntactic forms of a macro id
  - Pattern-based macros: the transformer associated with a macro id is defined by pattern matching (see next slide)

EXAMPLE: Let's define a macro with id `swap`, such that (`swap` *x* *y*) swaps the values of variables *x* and *y*.

- Intuition (`swap` *x* *y*) should expand to

```
(let ([tmp x])
     (set! x y)
     (set! y tmp))
```

$\Rightarrow$ the desired expansion rule is

```
(swap a b)  →  (let ([tmp a])
                    (set! a b)
                    (set! b tmp))
```

- An expansion rule has two parts: the pattern (left side) and the template (right side).
- *a* and *b* are called pattern variables. When matching (`swap` *x* *y*) with (`swap` *a* *b*), *a* is bound to *x* and *b* to *y*.

Pattern-based macros

Macros defined by a single pattern expansion

- `swap` is a macro whose expansion is described by a single pattern of the form (*macro-id* $pv_1 \ldots pv_n$) where $pv_1, \ldots, pv_n$ are distinct macro pattern variables.
- In RACKET, such a macro can be defined with

  (`define-syntax-rule` *pattern* *template*)

Example

```
(define-syntax-rule (swap a b)
  (let ([tmp a])
    (set! a b)
    (set! b tmp)))
```

Lexical scoping of macros

What should be the expansion of `(swap tmp other)`?

Naive expansion:

```
(let ([tmp tmp])
     (set! tmp other)
     (set! other tmp))
```

which keeps the bindings of `tmp` and `other` unchanged because:

- `tmp` is a lexically scoped variable of `let`, initialised with value 5 of global `tmp`
- `(set! tmp other)` changes the value of the local `tmp` to 5. The value of `tmp` is unaffected.
- `(set! other tmp)` reassigns to `other` its current value ⇒ the value of `other` is unaffected.

Lexical scoping of macros

- The naive expansion fails because of a *name clash*: `tmp` is both a pattern variable and in input variable.
- This problem can be avoided if all pattern variables which clash with input variables are renamed first ⇒ the clever expansion of `(swap tmp other)` is

```
(let ([tmp1 tmp])
     (set! tmp other)
     (set! other tmp1))
```

- GOOD NEWS: RACKET does clever expansion of macros.

We would like to apply `and` to a list of boolean values, but we can not:

```
> (apply and '(#t #f #f #t))  ; and is not a function
```

although the following expression yields the expected result:

```
> (and #t #f #f #t)  ; this is not a function call
#f
```

We can fix this problem, by defining a function that behaves like the macro, and use it instead of the macro:

```
> (define and-function
   (lambda xs
     (if (null? xs)
       #t
       (and (car xs) (apply and-function (cdr xs))))))
> (apply and-function '(#t #f #f #t))   ; ok
#f
```

Macros with multiple pattern expansions

RACKET allows to define macros whose expansions are given by different transformers for different patterns.

- Instead of `define-syntax-rule`, use `define-syntax` along with the `syntax-rules` transformer form:

  (`define-syntax` *pattern-id*
    (`syntax-rules` (id_1 ... id_m)
      [$pattern_1$ $template_1$]
      . . .
      [$pattern_n$ $template_n$]))

  - Different expansion rules [$pattern_i$ $template_i$] for different patterns.
  - $id_1, \ldots id_m$ are identifiers which should have no predefined meaning during the expansion phase.

A generalisation of `swap` intended to work with 2 or 3 variables:

```
> (define-syntax-rule (swap x y)
    (let ([tmp x])
      (set! x y)
      (set! y tmp)))
> (define-syntax rotate
    (syntax-rules ()
      ; transformer for first pattern of rotate
      [(rotate a b) (swap a b)]
      ; transformer for second pattern of rotate
      [(rotate a b c)
         (begin (swap a b) (swap b c))]))
> (let ([r 'v1] [g 'v2] [b 'v3])
      (rotate r g b) `(,r ,g ,b))
'(v2 v3 v1)
```

A generalisation of `swap` intended to work with any number of variables:

```
> (define-syntax rotate
    (syntax-rules ()
      ; transformer for pattern with 1 variable
      [(rotate a) (void)]
      ; transformer for pattern with at least 2 variables
      [(rotate a b c ...) (begin
                            (swap a b)
                            (rotate b c ...))]))
> (let ([x1 'v1] [x2 'v2] [x3 'v3] [x4 'v4])
      (rotate x1 x2 x3 x4) `(,x1 ,x2 ,x3 ,x4))
'(v2 v3 v4 v1)
```

- `c ...` indicates that `c` is a pattern variable for sequences of 0 or more forms.
- When a pattern variable like `c` is followed by `...` in a pattern, then it must be followed by `...` in the template, too.

Matching sequences in macro definitions

Towards a better implementation (1)

The previous implementation of `rotate` is slightly inefficient:

1. `rotate` of n variable performs $3 \cdot (n-1)$ `set!` operations.
2. $n+1$ `set!` operations are sufficient if (`rotate` x_1 ... x_n) has the expansion

   (`set! tmp` x_1) (`set!` x_1 x_2) ... (`set!` x_n `tmp`)

Implementation of this idea via a helper macro `shift-to`:

```
(define-syntax rotate
  (syntax-rules ()
    [(rotate a c ...)
     (shift-to (c ... a) (a c ...))]))
(define-syntax shift-to
  (syntax-rules ()
    [(shift-to (from0 from ...) (to0 to ...))
     (let ([tmp from0])
       (set! to from) ...
       (set! to0 tmp))]))
```

Special syntax for sequence of assignments:

- In the `shift-to` macro, `...` in the template follows `(set! to from)`, which causes the `(set! to from)` expression to be duplicated as many times as necessary to use each identifier matched in the `to` and `from` sequences.
  - The number of `to` and `from` matches must be the same, otherwise the macro expansion fails with an error.

Identifier macros

- The macros defined so far look like function names:

  (*macro-id* $form_1$ . . . $form_n$)

- Identifier macros look like identifiers: they have an associated transformer that replace them in-place with some other form.
  - Identifier macros are created with the combination of forms

```
(define-syntax pattern-id
  (syntax-id-rules (id_1 ... id_m)
    [pattern_1 template_1]
    ...
    [pattern_n template_n]))
```

Define an identifier macro `clock` such that

- `(set! clock` *form*`)` expands to `(put-clock` *form*`)`
- In all other contexts, `clock` expands to `(get-clock)`

```
(define-syntax clock
  (syntax-id-rules (set!)
    [(set! clock e) (put-clock! e)]
    [(clock a ...) ((get-clock) a ...)]
    [clock (get-clock)]))
```

- The expansion rules are tried top-down.
- The expansion rule for the patten `(clock a ...)` is needed because, when an identifier macro is used after an open parenthesis, the macro transformer is given the whole form, like with a non-identifier macro.

References

1. The RACKET Guide. Section 12.
2. The RACKET Guide. Section 16.