INFO0054-1
Programmation fonctionnelle

Quasiquotes and Pattern Matching

Christophe Debruyne

(c.debruyne@uliege.be)

Quotes

When evaluating `(quote e)`, the value that is returned is *e*, and not the value of *e*!

```
> (quote pi)        pi
> 'pi               pi
```

Not "useful" for booleans, strings and numbers as *e* and the value of *e* are the same.

```
> (equal? 3 '3)     #t
> (equal? + '+)     #f
```

It is useful for symbols (e.g., `'NOT`) and lists (e.g., `'(NOT a b)`), *of which the expressions would have been interpreted when not quoted*.

```
> (quote (+ 1 2))   (+ 1 2)
> '(+ 1 2)          (+ 1 2)
```

https://docs.racket-lang.org/guide/quote.html

Quasiquotes

Quasiquotes are similar to quotes but a quasiquote allows certain parts to be unquoted.

```
> (quote (+ 1 2 (+ 1 1 1)))                    (+ 1 2 (+ 1 1 1))
> '(+ 1 2 (+ 1 1 1))                           (+ 1 2 (+ 1 1 1))

> (quasiquote (+ 1 2 (+ 1 1 1)))               (+ 1 2 (+ 1 1 1))
> `(+ 1 2 (+ 1 1 1))                           (+ 1 2 (+ 1 1 1))

> (quasiquote (+ 1 2 (unquote (+ 1 1 1))))     (+ 1 2 3)
> `(+ 1 2 ,(+ 1 1 1))                          (+ 1 2 3)
```

https://docs.racket-lang.org/guide/qq.html

Quasiquotes


```
(define (n->var n) (string->symbol (format "x~a" n)))

(define (make-sum n)
  (cond [(= n 1) (n->var 1)]
        [else (quasiquote
          (+ (unquote (n->var n))
             (unquote (make-sum (- n 1)))))]))
```

```
> (n->var 2)
x2

> (make-sum 2)
(+ x2 x1)
```

Quasiquotes


```
(define (add-lets n body)
  (cond [(zero? n) body]
        [else
          (quasiquote
            (let ([(unquote (n->var n)) (unquote n)])
              (unquote (add-lets (- n 1) body))))]))

(define (build-exp n) (add-lets n (make-sum n)))
```

```
> (add-lets 2 'foo)
(let ((x2 2)) (let ((x1 1)) foo))

> (build-exp 2)
(let ((x2 2)) (let ((x1 1)) (+ x2 x1)))
```

Unquote-splicing

Unquote-splicing takes as argument a list and must be used where either a list or vector is expected.

In short: unquote-splicing evaluates the expression that leads to the list and then "trims" the outer parenthesis prior to placing the values in the quasiquote.

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

> `(1 2 (unquote-splicing (list (+ 1 2) (- 5 1))))
(1 2 3 4)
> `(1 2 ,@(list (+ 1 2) (- 5 1)))
(1 2 3 4)
```

Motivation for quotes and quasiquotes

Macros are functions that create code when interpreting the "static" source code. If you want to create abstractions for your source code, then macros are usually preferred. Macros are not covered in this course.

Together with `eval` (the function that takes a symbolic expression and evaluates it), quotes, quasiquotes and the symbolic expressions built with those forms provide a powerful mechanism to generate and evaluate *dynamic* code.

Pattern Matching
(filtrage par motif)

Matching symbolic expressions with a special form (and not the use of regular expressions for strings, numbers, etc.). With pattern matching, we can match arbitrary values.

Link cond, it tries to find *the first pattern* that matches the result of the value expression (input) and evaluates the corresponding bodies. Symbols in patterns act like variables that can be used in the body.

```
(define (f e)
  (match e
    ((list 'add a b) (+ a b))
    ((list 'subtract a b) (- a b))
    ((list 'divide a b) (/ a b))
    ((list 'multiply a b) (* a b))))
```

```
> (f '(add 4 5))
9
```

https://docs.racket-lang.org/guide/match.html
https://docs.racket-lang.org/reference/match.html

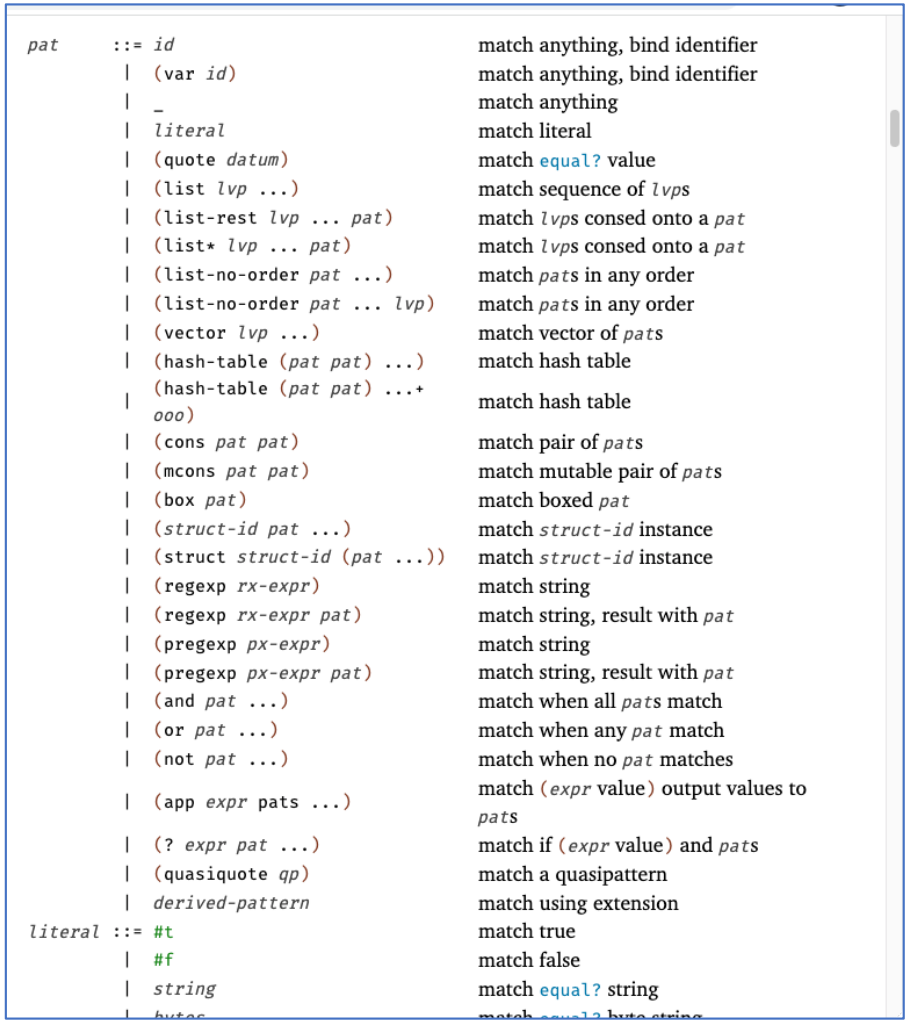
Pattern Matching


```
(define (f e)
  (match e
    ((list 'add a b) (+ a b))
    ((list 'subtract a b) (- a b))
    ((list 'divide a b) (/ a b))
    ((list 'multiply a b) (* a b))))
```

When no match is found, *an error is raised*:

```
> (f '(test 4 5))
match: no matching clause for (test 1 2)
```

Pattern Matching



pat	::= *id*	match anything, bind identifier
	| (var *id*)	match anything, bind identifier
	| _	match anything
	| *literal*	match literal
	| (quote *datum*)	match `equal?` value
	| (list *lvp* ...)	match sequence of *lvp*s
	| (list-rest *lvp* ... *pat*)	match *lvp*s consed onto a *pat*
	| (list* *lvp* ... *pat*)	match *lvp*s consed onto a *pat*
	| (list-no-order *pat* ...)	match *pat*s in any order
	| (list-no-order *pat* ... *lvp*)	match *pat*s in any order
	| (vector *lvp* ...)	match vector of *pat*s
	| (hash-table (*pat pat*) ...)	match hash table
	| (hash-table (*pat pat*) ...+ *ooo*)	match hash table
	| (cons *pat pat*)	match pair of *pat*s
	| (mcons *pat pat*)	match mutable pair of *pat*s
	| (box *pat*)	match boxed *pat*
	| (*struct-id pat* ...)	match *struct-id* instance
	| (struct *struct-id* (*pat* ...))	match *struct-id* instance
	| (regexp *rx-expr*)	match string
	| (regexp *rx-expr pat*)	match string, result with *pat*
	| (pregexp *px-expr*)	match string
	| (pregexp *px-expr pat*)	match string, result with *pat*
	| (and *pat* ...)	match when all *pat*s match
	| (or *pat* ...)	match when any *pat* match
	| (not *pat* ...)	match when no *pat* matches
	| (app *expr* pats ...)	match (*expr* value) output values to *pat*s
	| (? *expr pat* ...)	match if (*expr* value) and *pat*s
	| (quasiquote *qp*)	match a quasipattern
	| *derived-pattern*	match using extension
literal	::= #t	match true
	| #f	match false
	| *string*	match `equal?` string
	| *bytes*	match `equal?` byte string

Notice that some symbols are shared (e.g., list, cons, …). Those symbols are not referring to functions that are applied, but rather as part of the pattern.

There are also logical operators and patterns for quotes and quasiquotes.

Pattern Matching


```
(match val-expr clause ...)
1.      clause  =       [pat body ...+]
2.              |       [pat (=> id) body ...+]
3.              |       [pat #:when cond-expr body ...+]
```

We have already seen an example of (1). (2) uses failure procedures and may be interesting for those who want to look at that. We will see an example of (3) as it allows us, for instance, to have the same pattern with different conditions.

Pattern Matching


```
> (define (m x)
    (match x
      [(list a b c)
       #:when (= 6 (+ a b c))
       'sum-is-six]
      [(list a b c) 'sum-is-not-six]))
> (m '(1 2 3))
'sum-is-six
> (m '(2 3 4))
'sum-is-not-six
```

- "An optional #:when *cond-expr* specifies that the pattern should only match if *cond-expr* produces a true value."
- "*cond-expr* is in the scope of all of the variables bound in *pat*."
  - *This means it also has access to variables outside the match.*
- *"cond-expr* must not mutate the object being matched before calling the failure procedure, otherwise the behavior of matching is unpredictable."
  - *Why shouldn't this be an issue in functional programming?*

Pattern Matching

Pattern matching is available via the library `(require racket/match)`.

There are forms for:

- Matching all the values in a sequence `match*`
- Combining `let`s with pattern matching `match-let(*|rec)`
- …

These are techniques that are useful for your lab exercises and project. You may use these techniques during the tests and exam but will not be part of the those.