LispBM Cheatsheet

Basic Syntax

Comments

```
; Single line comment
```

Basic Values

```
42          ; Integer
3.14        ; 32-bit float
3.14f64     ; 64-bit double
"hello"     ; String
'symbol     ; Symbol
t           ; True
nil         ; False/empty list
```

Lists and Arrays

```
'(1 2 3)          ; Quoted list
(list 1 2 3)      ; Constructed list
[1 2 3]           ; Byte array
[| 1 2 3 |]       ; Lisp array
```

Core Functions

Arithmetic

```
(+ 1 2 3)         ; Addition: 6
(- 10 3)          ; Subtraction: 7
(* 2 3 4)         ; Multiplication: 24
(/ 12 3)          ; Division: 4
(// 7 3)          ; Integer division: 2
(mod 7 3)         ; Modulo: 1
```

Comparison

```
(= 5 5)           ; Numerical equality: t
(eq 'a 'a)        ; Symbol equality: t
(< 3 5)           ; Less than: t
(> 5 3)           ; Greater than: t
(<= 3 5)          ; Less or equal: t
(>= 5 3)          ; Greater or equal: t
```

List Operations

```
(car '(a b c))        ; First element: a
(cdr '(a b c))        ; Rest of list: (b c)
(cons 'x '(a b))      ; Prepend: (x a b)
(list 1 2 3)          ; Create list: (1 2 3)
(append '(1 2) '(3)) ; Join lists: (1 2 3)
(length '(a b c))     ; List length: 3
(ix '(a b c) 1)       ; Index access: b
```

Control Flow

Conditionals

```
(if (< x 10)
    "small"
    "big")

(cond
  ((< x 0) "negative")
  ((= x 0) "zero")
  (t "positive"))
```

Loops

```
(loop ((i 0))
      (< i 10)
      { (print i)
        (setq i (+ i 1)) })
```

Variable Binding

Local Variables

```
(let ((x 5) (y 10))
  (+ x y))

(var x 42)            ; Local binding in progn
{ (var x 42)
  (print x) }
```

Global Variables

```
(define x 42)         ; Global definition
(setq x 100)          ; Set variable
```

Functions

Function Definition

```
(define square (lambda (x) (* x x)))

(defun square (x) (* x x))      ; Alternative syntax
```

Function Calls

```
(square 5)            ; Call function: 25
(apply + '(1 2 3))    ; Apply function to list: 6
```

Expression Sequences

Sequential Evaluation

```
(progn expr1 expr2 expr3)     ; Evaluate in sequence
{ expr1 expr2 expr3 }         ; Brace shorthand
(atomic expr1 expr2 expr3)    ; Uninterruptible sequence
```

Pattern Matching

```
(match value
  (pattern1 result1)
  (pattern2 result2)
  (_ default-result))

; Pattern examples:
(match '(a b c)
  (((? x) (? y) (? z)) (list z y x))  ; Match exact structure
  (((? x) . (? xs)) x)                ; Match head/tail
  (_ 'no-match))
```

Concurrency

Process Creation

```
(spawn (lambda () (print "hello")))    ; Create process
(spawn-trap closure)                   ; Spawn with exit monitoring
(self)                                 ; Get current process ID
```

Message Passing

```
(send pid message)           ; Send message to process

(recv
  (pattern1 result1)         ; Receive and pattern match
  (pattern2 result2))

(recv-to timeout             ; Receive with timeout
  (pattern1 result1)
  (_ 'timeout))
```

Process Control

```
(yield 1000)                 ; Yield for microseconds
(sleep 1)                    ; Sleep for seconds
(wait pid)                   ; Wait for process to finish
(exit-ok value)              ; Exit successfully
(exit-error error)           ; Exit with error
```

Error Handling

```
(trap expression)            ; Catch errors
; Returns: '(exit-ok value) or '(exit-error error-symbol)

(define parse-number
  (lambda (str)
    (match (trap (str-to-i str))
      ('(exit-ok result) result)
      ('(exit-error _) 'invalid-number))))
```

Type Operations

```
(type-of value)              ; Get type of value
(list? value)                ; Test if list
(number? value)              ; Test if number
```

```
; Type conversion
(to-i 3.14)                 ; Convert to integer: 3
(to-float 42)               ; Convert to float: 42.0
```

Array Operations

```
; Byte arrays
(bufcreate 10)              ; Create 10-byte buffer
(bufget-u8 buf 0)           ; Get byte at index
(bufset-u8 buf 0 255)       ; Set byte at index

; Lisp arrays
(array 1 2 3)               ; Create array [| 1 2 3 |]
(array 1 'apa '(+ 1 2))     ; Create array [|1 apa (+ 1 2)|]
```

Advanced Features

Quasiquotation

```
`(list ,x ,@lst)            ; Quasiquote with unquote/splice
```

Macros

```
(define my-macro
  (macro (x) `(* ,x ,x)))
```

Continuations

```
(call-cc (lambda (k) ...))  ; Call with current continuation
```

Built-in Constants

```
nil                         ; Empty list / false
t                           ; True
```

Common Patterns

List Processing

```
; Map equivalent
(define map-square
  (lambda (lst)
    (if (eq lst nil)
        nil
        (cons (* (car lst) (car lst))
              (map-square (cdr lst))))))
```

Recursive Functions

```
(define factorial
  (lambda (n)
    (if (<= n 1)
        1
        (* n (factorial (- n 1))))))
```

Built-in Library Functions

Higher-Order Functions

```
(foldl + 0 '(1 2 3))          ; Fold left: 6
(foldr cons nil '(1 2 3))     ; Fold right: (1 2 3)
(filter (lambda (x) (> x 5)) '(1 6 3 8))  ; Filter: (6 8)
(zipwith + '(1 2 3) '(4 5 6))  ; Zip with function: (5 7 9)
(zip '(1 2 3) '(a b c))       ; Zip into pairs: ((1 . a) (2 . b) (3 . c))
```

List Utilities

```
(second '(a b c))             ; Second element: b
(third '(a b c))              ; Third element: c
(iota 5)                      ; Range 0 to n-1: (0 1 2 3 4)
```

Math Functions

```
(abs -5)                      ; Absolute value: 5
```

Array Utilities

```
(array? arr)                  ; Test if array: t/nil
(list-to-array '(1 2 3))      ; Convert list to array
(array-to-list arr)           ; Convert array to list
```

Advanced Macros

Alternative Function Definition

```
(defun square (x) (* x x))    ; Define function (alternative syntax)
(defunret factorial (n)       ; Function with early return capability
  (if (< n 0) (return 'error))
  (if (<= n 1) 1 (* n (factorial (- n 1)))))
```

Enhanced Loop Constructs

```
(loopfor i 0 (< i 10) (+ i 1)     ; For loop with iterator
  (print i))

(loopwhile (< i 10)               ; While loop
  { (print i) (setq i (+ i 1)) })

(looprange i 0 10                 ; Range-based loop
  (print i))

(loopforeach item '(a b c)        ; Foreach loop
  (print item))

(loopwhile-thd 1024 (< i 100)     ; Threaded while loop
  { (print i) (setq i (+ i 1)) })
```

Structure Definition

```
(defstruct point (x y))           ; Define struct type
; Creates:
; - (make-point x y)              ; Constructor
```

```
; - (point? obj)                ; Type predicate
; - (point-x obj)               ; Field accessor/setter
; - (point-y obj)               ; Field accessor/setter

(var p (make-point 10 20))      ; Create instance
(point-x p)                     ; Access field: 10
(point-x p 30)                  ; Set field
```

Advanced Macro Definition

```
(defmacro when (cond body)      ; Define macro
  `(if ,cond ,body nil))
```

Tips

- Use `=` for numbers, `eq` for symbols/booleans
- Brace syntax `{}` is shorthand for `progn`
- `trap` returns `'(exit-ok value)` or `'(exit-error symbol)`
- Process message passing enables Erlang-style concurrency
- Pattern matching works on lists, arrays, and values