Lisp Programming Cheatsheet

A concise reference for Lisp programming, covering syntax, data structures, functions, macros, and common operations. Useful for both beginners and experienced Lispers.

Basic Syntax & Data Types

Syntax Fundamentals

Lisp syntax is based on s-expressions (symbolic expressions).
Everything is either an atom or a list.

`(operator operand1 operand2 ...)` - Function application. The first element is the function, and the rest are its arguments.

Parentheses are crucial. They define the structure and order of operations.

Comments start with a semicolon `;` and continue to the end of the line.

Basic Data Types

Numbers:	Integers (e.g., `1`, `-42`), floating-point numbers (e.g., `3.14`, `-2.71`).
Symbols:	Represent variables, function names, etc. (e.g., `x`, `my-variable`). Case-insensitive by default (implementation dependent).
Strings:	Sequences of characters enclosed in double quotes (e.g., `"hello world"`).
Characters:	Represented differently depending on the Lisp dialect. (e.g., `#\A` in Common Lisp).
Booleans:	`t` (true) and `nil` (false). Note: `nil` also represents the empty list.
Lists:	Ordered collections of elements enclosed in parentheses (e.g., `(1 2 3)`, `(a b c)`).

Lists and Cons Cells

Lists are built from cons cells. A cons cell holds two pointers: `car` (first) and `cdr` (rest).

`cons` - Constructs a new cons cell.
`(cons 'a 'b)` ; => `(a . b)`

`car` - Returns the first element of a list.
`(car '(a b c))` ; => `a`

`cdr` - Returns the rest of the list (excluding the first element).
`(cdr '(a b c))` ; => `(b c)`

Functions and Control Flow

Defining Functions

`(defun function-name (parameter1 parameter2 ...) body)` - Defines a new function.

Example:

```
(defun square (x) (* x x))
(square 5) ; => 25
```

Parameters are symbols that receive the argument values when the function is called.

Control Flow

`if`	`(if condition then-clause else-clause)`
	Evaluates `condition`. If true, executes `then-clause`; otherwise, executes `else-clause`.
	`(if (> x 0) "positive" "non-positive")`

`cond`	`(cond (condition1 clause1) (condition2 clause2) ... (t else-clause))`
	A multi-way conditional. Evaluates conditions in order until one is true, then executes the corresponding clause.
	`(cond ((> x 0) "positive") ((< x 0) "negative") (t "zero"))`

`case`	A conditional that compares a key against multiple values. `(case x (1 "one") (2 "two") (otherwise "something else"))`
`loop` (Common Lisp)	Powerful iteration construct with many clauses for different looping behaviors. Too complex to summarize here, but essential for serious Lisp programming.

Lambda Functions

`(lambda (parameters) body)` - Creates an anonymous function.

```
((lambda (x) (* x x)) 5) ; => 25
```

Lambda functions are often used as arguments to other functions (higher-order functions).

`funcall` - Applies a function to arguments.
`(funcall #'+ 1 2)` ; => 3

`apply` - Applies a function to a list of arguments.
`(apply #'+ '(1 2))` ; => 3

Variables and Scope

Variable Binding

`let` - Introduces local variable bindings.
`(let ((variable1 value1) (variable2 value2) ...) body)`

```
(let ((x 10) (y 20))
  (+ x y))
; => 30
```

`let*` - Similar to `let`, but bindings are evaluated sequentially, allowing later bindings to depend on earlier ones.

```
(let* ((x 10)
       (y (+ x 5)))
  (* x y))
; => 150
```

`setf` - Assigns a value to a variable or a place.
`(setf variable value)`

```
(setf x 5)
x ; => 5
```

Scope

Lisp typically uses lexical (static) scoping. Variables are visible within the block they are defined and any nested blocks, unless shadowed by a new binding.

Global variables can be defined using `defvar` or `defparameter` (Common Lisp). `defparameter` is typically used for variables that you expect to change during program execution.

```
(defvar *global-variable* 10)
(defparameter *pi* 3.14159)
```

Data Structures

Arrays	Arrays can be created using `make-array` (Common Lisp). Access elements with `aref`.
	`(setf my-array (make-array '(3) :initial-contents '(1 2 3)))`
	`(aref my-array 0) ; => 1`

Hash Tables	Hash tables store key-value pairs. Created with `make-hash-table`. Access with `gethash` and set with `setf (gethash key hash-table) value`.
	`(setf my-hash (make-hash-table))`
	`(setf (gethash 'name my-hash) "Lisp")`
	`(gethash 'name my-hash) ; => "Lisp"`

Structures	User-defined data types with named slots. Defined using `defstruct` (Common Lisp).
	`(defstruct person name age)`
	`(setf p (make-person :name "Alice" :age 30))`
	`(person-name p) ; => "Alice"`

Macros

Macro Definition

`defmacro` - Defines a macro.
`(defmacro macro-name (parameters) body)`

Macros are code that write code. They are expanded at compile time.

Example:

```
(defmacro my-or (x y)
  `(let ((temp ,x))
     (if temp temp ,y)))
```

This macro defines a short-circuiting 'or' operator. `(my-or (print "x") (print "y"))` will only print "x" if x is not nil.

Quoting and Unquoting

`' (quote)`	Prevents evaluation. Returns the expression literally. `'(+ 1 2) ; => (+ 1 2)`
`` (backquote)`	Similar to `quote`, but allows selective evaluation using `,` (comma). `(let ((x 10))` `  `(the value of x is ,x))` `; => (the value of x is 10)`
`,` (comma)	Inside a backquote, evaluates the expression and splices the result. `(let ((numbers '(1 2 3)))` `  `(the numbers are ,@numbers))` `; => (the numbers are 1 2 3)`
,@	Used inside a backquote to splice a list into the surrounding list. `(let ((numbers '(1 2 3)))` `  `(numbers: ,@numbers))` `; => (numbers: 1 2 3)`

Macro Expansion

`macroexpand` - Shows the expanded form of a macro.

```
(macroexpand '(my-or (print "x") (print "y")))
; => (LET ((TEMP (PRINT "x")))
;      (IF TEMP TEMP (PRINT "y")))
```

Understanding macro expansion is crucial for debugging and understanding macro behavior.