Common Lisp Special Operators Cheat Sheet

◆ 束縛（6）

let	変数（並列束縛）	(let* ((x 1) (y (+ x 1))) (+ x y)) ; => 3	複数の値を複数のシンボル（変数）に束縛します。letは全ての変数を並列的に、let*は上から順に束縛します。 symbol-macroletは値ではなく式を束縛するため、無引数マクロとなります。
let*	変数（直列束縛）		
symbol-macrolet	シンボルマクロ		
flet	関数（再帰不可）	(flet ((double (x) (expt x 2)) (triple (x) (expt x 3))) (+ (double 5) (triple 5))) ; => 150	複数の関数を束縛します。fletは再帰を行わないので、同名の関数の呼出しは他の関数の呼出しを意味します。他方、labelsは再帰できます。 macroletはマクロを束縛します。 ※関数＝値を返す　／　マクロ＝式を返す
labels	関数（再帰可）		
macrolet	マクロ		

◆ 制御（3）

if	条件分岐	(if nil 1 0) ; => 0	引数がnilか否かで条件分岐を行います。
tagbody	タグ付ブロック	(tagbody (princ 0) (go t1) t2 (princ 2) (go out) t1 (princ 1) (go t2) out (princ 'end)) ; 012END	複数の式をタグ付でブロックにまとめます。タグ間の移動はgoで行うことができます。 最初の式にタグが付いていない場合は必ず実行されます。
go	ジャンプ		

◆ 固有機能（3）

function	関数実体の生成	#'print ; => <FUNCTION PRINT>	関数実体の取得や関数実体の生成をお細います。#'リーダーマクロを使えます。
quote	引用	'foo ; => foo	式や値をそのまま返します。
setq	変数の上書き	(setq foo 123) ; => 123	シンボルに値を設定（上書き）します。

◆ 宣言（2）

locally	ローカル宣言	(locally (declare …) …)	宣言を伴うブロック構造を導入します。
the	型宣言	(the <value-type> form)	局所的に型の宣言を行います。

◆ ブロック（4）

block	ブロックの導入	(block <name> body…)	複数式を名前付でまとめます。
return-from	ブロックからの脱出	(return-from <name> result)	名前付ブロックから脱出します。ブロック名がnilの場合はreturnマクロも使えます。
progn	複数式の実行	(progn body…)	単純に複数式を実行し、最後の式の返り値を返します。
progv	複数式の実行（束縛付）	(progv '(symbols…) '(values…) body…)	ダイナミック変数の束縛を伴って複数式を実行します。letはレキシカル変数ですが、progvはダイナミック変数であるため、シンボルのsymbol-valueスロットを操作します。

◆ 多値（2）

multiple-value-call	多値による関数呼出し	(multiple-value-call
		  #'function-form
		  multiple-value-forms…)
			関数に引数を適用するapplyに似ていますが、引数が多値の場合にも使用することができます。

multiple-value-prog1	多値を含む複数式の実行	(multiple-value-prog1 1st-form other-forms…)	複数式を実行しますが、返り値は最初の式の評価結果となります。

◆ 例外（3）

catch	例外脱出点の設定	(catch 'tag body…)	blockと同様に複数式をタグ付でまとめます。blockのnameはレキシカルスコープですが、catchのtagはダイナミックスコープです。主に例外脱出に用いられます。
throw	例外脱出	(throw 'tag result)	catchのbody内部で使用することで残りの式を評価せずに脱出します。

unwind-protect	例外脱出の無視	(unwind-protect
		  protected-form
		  cleanup-forms…)
			必ず実行したい式を記述します。第1引数の式を評価中に例外的な脱出が発生しても、まずはcleanup-formsを評価してから脱出します。

◆ ロード（2）

eval-when	評価時点の制御	(eval-when
		  (:compile-toplevel
		  :load-toplevel
		  :execute)
		  body…)
			通常の実行時だけでなくコンパイル時などにも実行します。マクロはコンパイル時に評価されている必要があるため、同一ファイルに記述する場合はeval-whenで括ります。

load-time-value	読込時評価	(load-time-value …)	LISPイメージのロード時に先行して評価されます。#.リーダーマクロは「リード時評価」ですが、こちらは「ロード時評価」であるため、LISPイメージが異なるとその都度評価されます。