ACL2 Language Constructs to Know - CSU290

Special expression forms

These are evaluated in special ways, unlike functions or function-like shorthand.

`(IF` *`test`* `  `*`true-part`* `  `*`false-part`*`)`	If *`test`* evaluates to `nil`, *`false-part`*; otherwise, *`true-part`*. (Other part is not evaluated.)

`(COND (`*`test1 expr1`*`)`
`  (`*`test2 expr2`*`)`
`  …`
`  (t `*`final-expr`*`))`
	Same as `(IF` *`test1 expr1`*
	`  (IF` *`test2 expr2`*
	`  …`
	`  `*`final-expr`*`))`

`(LET ((`*`var1 expr1`*`)`
`  (`*`var2 expr2`*`)`
`  …)`
`  `*`expr-body`*`)`
	Each *`exprk`* is evaluated and the result is bound to variable symbol *`vark`* in evaluation of *`expr-body`*.

`(LET* ((`*`var1 expr1`*`)`
`  (`*`var2 expr2`*`)`
`  …)`
`  `*`expr-body`*`)`
	Similar to `LET`, except that previous bindings in the same `LET*` can be used in later binding expressions in the same `LET*`.

`(QUOTE `*`value`*`)`	Evaluates to the literal value *`value`*. In other words, *`value`* is not evaluated as code.
`'`*`value`*	Shorthand for (QUOTE *`value`*)

Primitive functions

These are not defined in terms of other functions, and most of them cannot be. Thus, they are "primitive." (All functions described as "predicates" or "relations" return either `t` or `nil`).

`(EQUAL `*`a b`*`)`		Predicate for equality
`(CONS `*`a b`*`)`		Constructs a cons pair with CAR *`a`* and CDR *`b`*
`(CONSP `*`x`*`)`		Predicate for cons pairs
`(CAR `*`c`*`)`		The CAR of the cons *`c`* (or `nil`).
`(CDR `*`c`*`)`		The CDR of the cons *`c`* (or `nil`).
`(RATIONALP `*`x`*`)`		Predicate for rational numbers
`(NUMERATOR `*`r`*`)`		"Lowest terms" numerator of rational *`r`*
`(DENOMINATOR `*`r`*`)`		"Lowest terms" denominator of rational *`r`*
`(<` *`x y`*`)`		"Less than" relation on rationals
`(BINARY-*` *`x y`*`)`	(use * instead)	Multiplies two numbers
`(BINARY-+` *`x y`*`)`	(use + instead)	Adds two numbers
`(UNARY--` *`x`*`)`	(use – instead)	Negates a number, as in "zero minus ..."
`(UNARY-/` *`x`*`)`	(use / instead)	Inverts a number, as in "one over ..."
`(INTEGERP `*`x`*`)`		Predicate for integer numbers
`(STRINGP x)`		Predicate for strings
`(SYMBOLP x)`		Predicate for symbols
`(CHARACTERP x)`		Predicate for characters

Built-in Shorthand

These are used extensively in ACL2 and behave much like functions. You can pretend they are functions that, in many cases, take a variable number of arguments.

Arithmetic	
(+ …)	Sum of all the parameters
(* …)	Product of all the parameters
(- *x*)	Negates a number, as in “zero minus …”
(- *x y*)	Subtracts number *y* from number *x*
(/ *x*)	Inverts a number, as in “one over …”
(/ *x y*)	Divides number *x* by number *y*
(<= *x y*)	(not (< *y x*))
(> *x y*)	(< *y x*)
(>= *x y*)	(not (< *x y*))
Booleans	
(AND …)	Conjunction of all the parameters
(OR …)	Disjunction of all the parameters
Lists	
(LIST …)	Constructs a list of all the parameters
(APPEND *a b* …)	Appends all of the parameter lists together

Built-in Derived Functions

These are defined in terms of primitive functions, special forms, and shorthand. You can use **:**PE *name* to display the actual definition.

Arithmetic	
(NATP *x*)	Predicate for natural numbers (integers >= 0)
(ZP *x*)	Predicate for [x = 0 or x is not a natural]
(ZIP *x*)	Predicate for [x = 0 or x is not an integer]
(= *x y*)	Same as equal; intended for numbers
Booleans	
(IMPLIES *p q*)	Logical/boolean implication
(NOT *p*)	Logical/boolean negation
(IFF *p q*)	Logical/boolean equivalence
Lists	
(ATOM *x*)	Predicate for non-conses
(ENDP *x*)	Same as atom; intended for lists
(TRUE-LISTP *x*)	Predicate for nil-terminated lists
(LEN *x*)	Length of a list
(BINARY-APPEND *x y*) (use append instead)	Returns the list with *x* appended before *y*