HISTORY: The following information is derived from the history section of dpANS Common Lisp. Lisp is a family of languages with a long history. Early key ideas in Lisp were developed by John McCarthy, during the 1956 Dartmouth Summer Research Project on Artificial Intelligence. McCarthy's motivation was to develop an algebraic list processing language for artificial intelligence work. Implementation efforts for early dialects of Lisp were undertaken on the IBM 704, the IBM 7090, the Digital Equipment Corporation (DEC) PDP-1, the DEC PDP-6, and the PDP-10. The primary dialect of Lisp between 1960 and 1965 was Lisp 1.5. By the early 1970's there were two predominant dialects of Lisp, both arising from these early efforts: MacLisp and Interlisp. MacLisp improved on the Lisp 1.5 notion of special variables and error handling. MacLisp also introduced the concept of functions that could take a variable number of arguments, macros, arrays, non-local dynamic exits, fast arithmetic, the first good Lisp compiler, and an emphasis on execution speed. Interlisp introduced many ideas into Lisp programming environments and methodology. One of the Interlisp ideas that influenced Common Lisp was an iteration construct implemented by Warren Teitelman that inspired the loop macro used both on the Lisp Machines and in MacLisp, and now in Common Lisp. One of the most important developments in Lisp occurred during the second half of the 1970's: Scheme. Scheme, designed by Gerald J. Sussman and Guy L. Steele Jr., is a simple dialect of Lisp whose design brought to Lisp some of the ideas from programming language semantics developed in the 1960's. In the late 1970's object-oriented programming concepts started to make a strong impact on Lisp. At MIT, certain ideas from Smalltalk made their way into several widely used programming systems. Flavors, an object-oriented programming system with multiple inheritance was developed at MIT for the Lisp machine community by Howard Cannon and others. At Xerox, the experience with Smalltalk and Knowledge Representation Language (KRL) led to the development of Lisp Object Oriented Programming System (LOOPS) and later Common LOOPS. These systems influenced the design of the Common Lisp Object System (CLOS). CLOS was developed specifically for X3J13's standardization effort, and was separately written up in common object oriented Lisp systems specifications. However, minor details of its design have changed slightly since that publication. In April 1981,
after a DARPA-sponsored meeting concerning the splintered Lisp community, Symbolics, the SPICE project, the NIL project, and the S-1 Lisp project joined together to define Common Lisp. Initially spearheaded by White and Gabriel, the driving force behind this grassroots effort was provided by Fahlman, Daniel Weinreb, David Moon, Steele, and Gabrie. In 1986 X3J13 was formed as a technical working group to produce a draft for an ANSI Common Lisp standard. Because of the acceptance of Common Lisp, the goals of this group differed from those of the original designers. These new goals included stricter standardization for portability, an object-oriented programming system, a condition system, iteration facilities, and a way to handle large character sets. To accommodate those goals, a new language specification was developed.

**CLOS (Common Lisp Object System)** is a dynamic, object oriented extension to Common Lisp that provides a powerful general purpose programming language. It is a popular tool for development of large scale systems. The value of CLOS is becoming more widely recognized today, based on its features and its very modern object system approach, supporting features such as the following:

1. Support for modules (or packages), preventing name clashes
2. A unified syntax derived from Common Lisp:
   a. A multiple inheritance scheme
   b. Generic function calls and message passing
   c. A combination of methods related to function composition
   d. Class re-definitions and updating
   e. Relationships between generic functions through a "meta object protocol"

Integration with Windows and with native operating systems and with other languages such as C is straightforward. CLOS offers an "open world" approach to systems development, and supports optimization for speed. CLOS is a very modern dynamic, principled, object-oriented, class-centered, method-centered approach to tailoring a system. It offers a major benefit of reducing the turn-around time for development. This means you can get your applications to market faster, and often at lower programming costs. It also means it is faster and easier to make changes to your applications for example, to
provide upgrades. For example, in the course of debugging a system you can look and realize you didn't write a class correctly. So right on the spot you can say "redefine the class" and it all happens dynamically--the system adjusts on the fly. You don't have to go back to ground zero and recompile all the dependency chains that are dependent on that class definition and so on (there could be hundreds or thousands of them).

The whole point of the object oriented approach is to take something and to customize it without having to change the code, without having to go back and edit any of the lines of code that already exist. Objectivity allows someone who doesn't have access to your source code to extend the functionality of functions that you have written without being forced to become involved in the details of the program you provided them. Users don't have to change their source code to use your extensions. This opens up the use of whole worlds of off-the-shelf class libraries of functionality.

CLOS also offers the capability of enabling your instances (or objects in general) to live from session to session. This powerful capability enables you to have your objects "persist" so that they may be returned as the result of a function or procedural call. The object may be the value of a function parameter or returned as a functional value. This capability allows you to make subclasses of a standard class that has the behavior you desire. It also enables your LISP functions to create new functions at runtime or alter old ones, depending on other parameters, saving significant amounts of programming time.

Some developers feel that Smalltalk's object model is too restrictive, making system optimization quite difficult in many applications. For example, Smalltalk lacks multiple inheritance, which CLOS offers. CLOS also offers multi-methods (or Mixins), which enable a much more concise expression of problems, which in turn is very important to incremental development. Smalltalk's GUI framework dates back long before current GUI approaches, which can make it difficult to integrate Smalltalk into current GUI approaches. These are examples of why some experienced developers feel LISP with CLOS is a significantly more robust environment than other object oriented languages can offer. Because of the standards that exist for CLOS, it is much easier to port CLOS code than when using code from less mature object oriented languages.
It is also interesting to note that a functional language (such as LISP with CLOS) is inherently well suited for parallel processing. If a function is free of side effects, values for the various variables can be computed independently and simultaneously. A programmer using a functional language need not give extraordinary thought to parallelism but simply program functions. A smart compiler will pick out which parameters should be evaluated in parallel.

Thousands of new users are beginning to experience for the first time the benefits of the power and flexibility offered by dynamic object oriented programming environments enhanced by the addition of CLOS.

This program simply demonstrates the return of a string from a function call, in the Lisp programming language. The program will display the message "Hello world" each time the function **HELLO** is called.

Source Code

```lisp
;; HWorld.lsp

;;; ================================================== ;;;
;;; =========== HELLO WORLD SIMULATION ============== ;;;
;;; ================================================== ;;;

;;; This function simply returns the string Hello World that is in quotes.
(DEFUN HELLO ()
   "HELLO WORLD"
)

WORKS CITED

The main purpose of this program was to implement a Craps game, using a language that we have just learned. Also, it was written in a functional style with almost no reliance on the assignment operation. Only one local variable called THROW was used:

**CRAPS SIMULATION**

**This function takes no parameters as input and returns a random number between 1 and 6.**

```lisp
(DEFUN THROW-DIE ()
  (+ (RANDOM 6) 1) ;;; get a random number between 0 and 5 and then add 1
)
```

**This function takes no parameters as input and returns a LIST with two numbers between 1 and 6.**

```lisp
(DEFUN THROW-DICE ()
  (LIST (THROW-DIE) (THROW-DIE)) ;;; create a list with two random numbers
)
```
;;; ** This function takes two numbers as parameters for input and returns T or Nil. T is returned if both numbers are equal to 6. Nil is returned otherwise. **

(DEFUN BOXCARS-P (A B)
  (AND (EQUAL '6 A)
    (EQUAL '6 B))
)

;;; ** This function takes two numbers as parameters for input and returns T or Nil. T is returned if both numbers are equal to 1. Nil is returned otherwise. **

(DEFUN SNAKE-EYES-P (A B)
  (AND (EQUAL '1 A)
    (EQUAL '1 B))
)

;;; ** This function takes two numbers as parameters for input and returns T or Nil. T is
;;; returned if the sum of both numbers is equal to a 7 or 11. Nil is returned otherwise. **

(DEFUN INSTANT-WIN-P (A B)
    (OR (EQUAL '7 (+ A B))
        (EQUAL '11 (+ A B)))
)

;;; ** This function takes two numbers as parameters for input and returns T or Nil. T is
;;; returned if the sum of both numbers is equal to a 2, 3 or 12. Nil is returned otherwise. **

(DEFUN INSTANT-LOSS-P (A B)
    (OR (EQUAL '2 (+ A B))
        (EQUAL '3 (+ A B))
        (EQUAL '12 (+ A B)))
)
(DEFUN SAY-THROW (A B)
  (COND ((BOXCARS-P A B) 'BOXCARS) ;;; make use of function
        (SNAKE-EYES-P A B) 'SNAKE-EYES) ;;; make use of function
  ))

;;; ** This is the main function used to simulate the game of craps. Variable THROW
;;; contains a LIST of two;;; numbers between 1 and 6. The numbers located in THROW,
;;; are used as parameters for the other functions.;;; The several pieces used for output
;;; are listed together and then the LIST is returned from this ;;; function.

(DEFUN CRAPS ()
  )
)
(LET THROW (THROW-DICE)) ;; get initial roll of the dice

;; if roll is a win, then LIST the appropriate output

(COND ((INSTANT-WIN-P (FIRST THROW) (SECOND THROW))

(LIST 'THROW (FIRST THROW) 'AND (SECOND THROW) '-- (+ (FIRST
THROW) (SECOND THROW)) '-- 'YOU 'WIN)) ;; if roll is a loss, then check for

BOXCARS or SNAKE-EYES

((INSTANT-LOSS-P (FIRST THROW) (SECOND THROW))

(IF (EQUAL 'NIL (SAY-THROW (FIRST THROW) (SECOND THROW))) ;;
if Nil then LIST appropriate output

(LIST 'THROW (FIRST THROW) 'AND (SECOND THROW) '-- (+ (FIRST
THROW) (SECOND THROW)) '-- 'YOU 'LOSE) ;; else include the BOXCARS or

SNAKE-EYES string in the output

(LIST 'THROW (FIRST THROW) 'AND (SECOND THROW) '-- (SAY-
THROW (FIRST THROW) (SECOND THROW))

'-- 'YOU 'LOSE)))

;; if roll is not instant win or loss then output sum of dice

(T (LIST 'THROW (FIRST THROW) 'AND (SECOND THROW) '-- 'YOUR

'POINT 'IS (+ (FIRST THROW)
(SECOND THROW))))

) ;; end COND

) ;; end LET

)
END OF PROGRAM

CRAPS

;;;