CLIPS Architecture Manual

CLIPS Version 5.1

January 6th 1992

Software Technology Branch
Lyndon B. Johnson Space Center
CONTENTS

Preface ................................................................................................................... i
Acknowledgements ............................................................................................... v
Introduction ............................................................................................................ 1
System Dependent Module .................................................................................... 5
Memory Module ..................................................................................................... 7
Symbol Manager Module ....................................................................................... 17
Router Module ....................................................................................................... 27
Scanner Module ..................................................................................................... 35
Expression Module ................................................................................................. 41
Special Forms Module ............................................................................................ 49
Parser Utility Module ............................................................................................. 53
Evaluation Module .................................................................................................. 57
Command Line Module ........................................................................................... 63
Construct Manager Module .................................................................................... 69
Utility Module ......................................................................................................... 75
Fact Manager Module ............................................................................................. 85
Fact Commands Module ......................................................................................... 93
Deffacts Manager Module ......................................................................................... 95
Defglobal Manager Module .................................................................................... 101
Defrule Parser Module ......................................................................................... 109
Reorder Module ..................................................................................................... 117
The History of CLIPS

The origins of the C Language Integrated Production System (CLIPS) date back to 1984 at NASA’s Johnson Space Center. At this time, the Artificial Intelligence Section (now the Software Technology Branch) had developed over a dozen prototype expert systems applications using state-of-the-art hardware and software. However, despite extensive demonstrations of the potential of expert systems, few of these applications were put into regular use. This failure to provide expert systems technology within NASA’s operational computing constraints could largely be traced to the use of LISP as the base language for nearly all expert system software tools at that time. In particular, three problems hindered the use of LISP based expert system tools within NASA: the low availability of LISP on a wide variety of conventional computers, the high cost of state-of-the-art LISP tools and hardware, and the poor integration of LISP with other languages (making embedded applications difficult).

The Artificial Intelligence Section felt that the use of a conventional language, such as C, would eliminate most of these problems, and initially looked to the expert system tool vendors to provide an expert system tool written using a conventional language. Although a number of tool vendors started converting their tools to run in C, the cost of each tool was still very high, most were restricted to a small variety of computers, and the projected availability times were discouraging. To meet all of its needs in a timely and cost effective manner, it became evident that the Artificial Intelligence Section would have to develop its own C based expert system tool.

The prototype version of CLIPS was developed in the spring of 1985 in a little over two months. Particular attention was given to making the tool compatible with expert systems under development at that time by the Artificial Intelligence Section. Thus, the syntax of CLIPS was made to very closely resemble the syntax of a subset of the ART expert system tool developed by Inference Corporation. Although originally modelled from ART, CLIPS was developed entirely without assistance from Inference or access to the ART source code.

The original intent of the prototype was to gain useful insight and knowledge about the construction of expert system tools and to lay the groundwork for the construction of a fully usable tool. The CLIPS prototype had numerous shortcomings, however, it demonstrated the feasibility of the project concept. After additional development, it became apparent that sufficient enhancements to the prototype would produce a low cost expert system tool that would be ideal for the purposes of training. Another year of development and internal use went into CLIPS improving its portability, performance, and functionality. A reference manual and user’s guide were written during this time.
The first release of CLIPS to groups outside of NASA, version 3.0, occurred in the summer of 1986.

Further enhancements transformed CLIPS from a training tool into a tool useful for the development and delivery of expert systems as well. Versions 4.0 and 4.1 of CLIPS, released respectively in the summer and fall of 1987, featured greatly improved performance, external language integration, and delivery capabilities. Version 4.2 of CLIPS, released in the summer of 1988, was a complete rewrite of CLIPS for code modularity. Also included with this release were an architecture manual providing a detailed description of the CLIPS software architecture and a utility program for aiding in the verification and validation of rule-based programs. Version 4.3 of CLIPS, released in the summer of 1989, added still more functionality.

Originally, the primary representation methodology in CLIPS was a forward chaining rule language based on the Rete algorithm (hence the Production System part of the CLIPS acronym). Version 5.0 of CLIPS, released in the spring of 1991, introduced two new programming paradigms: procedural programming (as found in languages such as C and Ada) and object-oriented programming (as found in languages such as the Common Lisp Object System and Smalltalk). The object-oriented programming language provided within CLIPS is called the CLIPS Object-Oriented Language (COOL).

Because of its portability, extensibility, capabilities, and low-cost, CLIPS has received widespread acceptance throughout the government, industry, and academia. The development of CLIPS has helped to improve the ability to deliver expert system technology throughout the public and private sectors for a wide range of applications and diverse computing environments. CLIPS is being used by over 3,300 users throughout the public and private community including: all NASA sites and branches of the military, numerous federal bureaus, government contractors, 170 universities, and many companies. CLIPS is available at a nominal cost through COSMIC, the NASA software distribution center (for more on COSMIC, see appendix E of the Basic Programming Guide).

**CLIPS Version 5.1**

Version 5.1 of CLIPS is primarily a software maintenance upgrade required to support the newly developed and/or enhanced X Window, MS-DOS, and Macintosh interfaces. For a detailed listing of differences between versions 4.3, 5.0, and 5.1 of CLIPS, refer to appendix D of the Basic Programming Guide.

**CLIPS Documentation**

Three documents are provided with CLIPS.
• The CLIPS Reference Manual which is split into the following parts:
  
  • Volume I - The Basic Programming Guide, which provides the definitive description of CLIPS syntax and examples of usage.
  
  • Volume II - The Advanced Programming Guide, which provides detailed discussions of the more sophisticated features in CLIPS and is intended for people with extensive programming experience who are using CLIPS for advanced applications.
  
  • Volume III - The Utilities and Interfaces Guide, which provides information on machine-specific interfaces and CLIPS utility programs.
  
• The CLIPS User's Guide which provides an introduction to CLIPS and is intended for people with little or no expert system experience.
  
  • Volume I - Rules, which provides an introduction to rule-based programming using CLIPS.
  
  • Volume II - Objects, which provides an introduction to object-oriented programming using COOL.
  
• The CLIPS Architecture Manual which provides a detailed description of the CLIPS software architecture. This manual describes each module of CLIPS in terms of functionality and purpose. It is intended for people with extensive programming experience who are interested in modifying CLIPS or who want to gain a deeper understanding of how CLIPS works.
As with any large project, CLIPS is the result of the efforts of numerous people. The primary contributors have been: Robert Savely, head of the STB, who conceived the project and provided overall direction and support; Frank Lopez, who wrote the original prototype version of CLIPS; Gary Riley, who rewrote the prototype and is responsible for most of the kernel code; Chris Culbert, who managed the project, wrote the original CLIPS Reference Manual, and designed the original version of CRSV; Dr. Joseph Giarratano of the University of Houston-Clear Lake, who wrote the CLIPS User's Guide; Brian Donnell, who designed and developed the CLIPS Object Oriented Language (COOL); and Bebe Ly, who is responsible for maintenance and enhancements to CRSV.

Introduction

This manual provides an architecture description for version 5.0 of CLIPS. Each module of the CLIPS program is described in terms of its functionality and purpose. In addition, significant variables and functions (both local and global to the modules) are described. All functions relating to a given module are not necessarily listed. In other cases, some function names may not directly correspond to their counterpart in the CLIPS source code. This manual is intended partly as a set of instructions for building CLIPS from scratch and partly as a roadmap to the `C` implementation of CLIPS.

Function and variable names will be shown in boldface when they are referred to in a sentence. Other words which may cause confusion when used in a sentence will also be shown in boldface. For example, the word `and` can refer either to the function `and` or to the conditional element `and`.

This manual is written with the assumption that the reader has a basic understanding of the Rete Match Algorithm. A good source for information on the Rete Match Algorithm is Charles Forgy's Ph.D. Dissertation, “On the Efficient Implementation of Production Systems.” It can be obtained from University Microfilms International

300 N. Zeeb Road
Ann Arbor, MI 48106
(313) 761-4700

Another source for information is Charles Forgy's article “Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.” This can be found in Artificial Intelligence 19, pp. 17-37, 1982.

Document Overview

The modules described in this document are listed in order beginning with the lower-level modules and ending with the higher-level modules. The higher-level modules generally require the lower-level modules to operate.

The first four modules (System Dependent, Memory, Symbol Manager, and Router) provide basic support for very low-level CLIPS operations. The System Dependent Module (sysdep.c) implements system-dependent features such as timing functions. The Memory Module (memory.c) is used to efficiently allocate and maintain memory requests. The Symbol Manager Module (symbol.c) is used to avoid storage duplication for multiple occurrences of symbols, floats, and integers. It also assures that storage is not used for symbols, floats, and integers that are no longer in use. The Router Module (router.c) handles input/output (I/O) requests and allows these requests to be redirected to different I/O handlers. This redirection capability allows sophisticated interfaces to be built on top of the CLIPS kernel without making changes to the code.

The next eight modules (Scanner, Expression, Special Forms, Parser Utility, Evaluation, Command Line, Construct Manager, and Utility) provide the basic functionality necessary for expression evaluation, construct support, and the CLIPS command line interface. The Scanner Module (scanner.c) reads tokens from an input source. The Expression Module (expressn.c) builds expressions from tokens returned by the Scanner Module. The Special Forms Module (spclform.c) is used for parsing...
functions that do not conform with the standard syntax for function expressions (such as the `assert` function). The Parser Utility Module (parsutil.c) contains some utility functions useful for parsing both functions and constructs. The Evaluation Module (evaluatn.c) can evaluate expressions generated by the Expression Module. The Command Line Module (commline.c) provides the necessary functionality for a command line interface. It also is capable of determining when an expression has been formed from a series of input characters before it calls the Expression Module to build an expression. It then calls the Evaluation Module to evaluate the expression. The Construct Manager Module (constrct.c) provides the necessary support for registering constructs so that they are recognized by the CLIPS parser. It calls the appropriate routines needed by each construct for loading and parsing, resetting, and clearing. The Utility Module (utility.c) provides a number of general purpose routines for printing values, detecting errors, handling garbage collection, and registering items for use with the `watch` command.

The Fact Manager Module (factmngr.c) is used to maintain the fact-list and provide support for the creation of multifield values (by functions such as `mv-append`). The Fact Commands Module (factcom.c) implements the top level interface for commands such as `assert` and `facts`.

The Deffacts Module (deffacts.c) provides the capability needed to implement the deffacts construct. The Defglobal Module (defglobl.c) provides the capability needed to implement the defglobal construct.

The next six modules (Defrule Parser, Reorder, Variable Manager, Analysis, Generate, and Build) are used to build the appropriate data structures for the defrule construct. The Defrule Parser Module (ruleprsr.c) is used to parse the left-hand side (LHS) of a rule, yielding an intermediate data structure. The Reorder Module (reorder.c) transforms the LHS of a single rule containing `and` and `or` conditional elements nested throughout the intermediate LHS data structure into an intermediate data structure which contains at most a single `or` conditional element at the beginning of the intermediate data structure. The Variable Manager Module (variable.c) checks the patterns on the LHS of a rule for semantic errors involving variables. It also maintains information about the location and usage of variables in the patterns of a rule. The Analysis Module (analysis.c) works closely with the Variable Manager Module to generate expressions for the rule that will be used in the join and pattern networks. The Generate Module (generate.c) is used to generate the expressions requested by the Analysis Module. The Build Module (build.c) is used to integrate the new rule and its expressions into the join and pattern network.

The next six modules (Drive, Engine, Match, Retract, Rete Utility, and Logical Dependencies) form the core of the CLIPS inference engine. The Drive Module (drive.c) is used to update the join network when a fact has been added. The Engine Module (engine.c) maintains the agenda and handles execution of the RHS of rules. The Match Module (match.c) determines which patterns in the pattern network have been matched when a fact has been added. The Retract Module (retract.c) is used to update the join network when a fact is removed. The Rete Utility Module (reteutil.c) provides useful utility functions used by other modules for maintaining the join network. The Logical Dependencies Module (lgcldpnd.c) is used to maintain the links between the join network and facts to support the `logical` conditional element.

The Defrule Manager Module (defrule.c) coordinates the activities of all modules used for maintaining the defrule construct. The Defrule Deployment Module (drulebin.c) provides the functionality needed to use the defrule construct with the
bsave, bload, and constructs-to-c commands. The Defrule Commands Module (rulecom.c) implements the top level interface for defrule commands. The Deftemplate Command Module (deftmcom.c) is used for maintaining deftemplates, providing type and value checking for deftemplate slots, providing the top level interface for deftemplate commands, and providing the functionality needed to use the deftemplate construct with the bsave, bload, and constructs-to-c commands. The Deftemplate Function Module (deftmfun.c) is used for parsing assert, modify, and duplicate commands which use deftemplate formats. The Deftemplate Parser Module (deftmpsr.c) is used to parse the deftemplate construct. The Deftemplate LHS Module (defmlhs.c) is used to parse deftemplate patterns found on the LHS of a rule.

The Binary Save Module (bsave.c) provides the functionality needed for the bsave command, the Binary Load Module (bload.c) provides the functionality needed for the bload command, and the Construct Compiler Module (constrct.c) provides the functionality needed for the constructs-to-c command.

The next nine modules (Primary Functions, Predicate Functions, I/O Functions, Secondary Functions, Multifield Functions, String Functions, Math Functions, Text Processing Functions, and File Commands) provide functions and commands for a variety of tasks. The Primary Functions Module (sysprime.c) provides a set of environment commands and procedural functions. The Predicate Functions Module (syspred.c) provides a number of predicates and simple mathematical functions commonly used in CLIPS. The I/O Functions Module (sysio.c) provides a number of functions convenient for performing I/O. The Secondary Functions Module (syssecnd.c) provides a set of useful functions that perform a wide variety of useful tasks. The Multifield Functions Module (multivar.c) provides a set of useful functions for use with multifield values. The String Functions Module (strings.c) provides a set of useful functions for manipulating strings. The Math Functions Module (math.c) provides a set of useful math functions beyond the basic math functions provided by the Predicate Functions Module. The Text Processing Module (textpro.c) provides a set of useful functions for building and accessing a hierarchical lookup system for multiple external files. The File Commands Module (filecom.c) provides a set of useful interface commands that performs certain file operations not associated with standard file I/O operations.

The Deffunction Module (deffnctn.c) provides the capability to define new user-defined functions directly in CLIPS.

The next four modules implement overloaded functions which can be defined directly in CLIPS: Generic Function Commands, Generic Function Functions, Generic Function Construct Compiler Interface and Generic Function Binary Load/Save Interface. Generic functions can do different things depending on the number and type of arguments they receive. The Generic Function Commands Module (genrccom.c) contains most of the parsing routines necessary for generic functions and their methods. The Generic Function Functions Module (genrcfun.c) determines the precedence between different methods of a generic function, provides the generic dispatch when a generic function is actually called and contains various other maintenance routines for generic functions and their methods. The Generic Function Construct Compiler Interface (genrccmp.c) and the Generic Function Binary Load/Save Interface Modules provide the interfaces for generic functions to the constructs-to-c and bload/bsave commands.
The next ten modules give all the functionality of the CLIPS Object-Oriented Language (COOL): Class Commands, Class Functions, Instance Commands, Instance Functions, Message-Handler Commands, Message-Handler Functions, Instance-Set Queries, Definstances, Object Construct Compiler Interface and Object Binary Load/Save Interface. The Class Commands Module (classcom.c) furnishes the parsing and general interface routines for the defclass construct. The Class Functions Module (classfun.c) handles all the internal manipulations of classes, including the construction of class precedence lists from multiple inheritance. The Instance Commands Module (inscom.c) provides the parsing and general interface functions for instances of user-defined classes. The Instance Functions Module (insfun.c) deals with the internal details of creating, accessing and deleting instances. The Message-Handler Commands Module (msgcom.c) contains the parsing and general interface routines for the procedural attachments to classes. The Message-Handler Functions Module (msgfun.c) implements the message dispatch when a message is actually sent to an object and maintains the internal details of the defmessage-handler construct. The Instance-Set Queries Module (insquery.c) provides the routines for a useful query system which can determine and perform actions on sets of instances of user-defined classes that satisfy user-defined criteria. The Definstances Module (defins.c) provides the capability needed to implement the definstances construct. The Object Construct Compiler Interface (objcmp.c) and the Object Binary Load/Save Interface (objbin.c) Modules provide the interfaces for COOL to the constructs-to-c and bload/bsave commands.

The Main Module (main.c) contains the CLIPS startup function and should be the only file modified to add extensions or to embed CLIPS under normal circumstances.

Portability Notes

There are a number of coding practices in the CLIPS code that in general have proven to be portable among a wide variety of machines, but that are not guaranteed to be portable for all ANSI C compilers. In particular, the conversion of integers to pointers (and conversion back again expecting the original integer) is used quite extensively to implement the bload/bsave commands. Strict ANSI C conformance does not guarantee the portability of converting non-zero integers to pointers. Such a conversion may involve a representation change which would cause a subsequent conversion back to an integer to yield a value other than the original starting integer. Some machines may also generate access violations when attempting to store integers into pointer values when the integers represent invalid addresses. Also, similar to typecasting integers to pointers, typecasting a pointer type into another pointer type and expecting to be able to retrieve the original pointer is also not always ANSI C conformant (depending upon the pointer types). The code may be changed to be more portable in the next release. Compromises to functionality and efficiency will be considered when making these determinations.
System Dependent Module

The System Dependent Module (sysdep.c) maintains a set of functions that contains system and/or machine dependent features (such as timing functions) and initialization routines. The generic setting for CLIPS will compile the functions in this module to forms which should run on any system or machine.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

None.

GLOBAL FUNCTIONS

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>Arguments</th>
<th>C Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CatchControlC</td>
<td>A function which provides for interrupt handling. Used to break execution when ctrl-C is pressed.</td>
<td></td>
<td>Handled on most machines by using the <code>signal</code> function.</td>
</tr>
</tbody>
</table>
C IMPLEMENTATION: Handled on most machines by using the `srand` function.

**gensystem**

PURPOSE: A generic function which access to Operating System (OS) commands.

ARGUMENTS: A string which is a command to be executed by the OS.

**gentime**

PURPOSE: A generic function for providing time information.

RETURNS: Current time as a floating-point number.

**InitializeCLIPS**

PURPOSE: Performs initialization of CLIPS.

OTHER NOTES: Initialization differs between standard and run-time configurations.

**InitializeNonportableFeatures**

PURPOSE: Performs machine-dependent initialization for features such as interrupt handling.

**RerouteStdin**

PURPOSE: Forces CLIPS to read input from a file when the `-f` option is used when CLIPS is first started.

**SystemFunctionDefinitions**

PURPOSE: Sets up the definitions of CLIPS system defined functions.
Memory Module

Allocation of memory occurs constantly during loading, browsing, and execution of CLIPS programs. Memory allocation/deallocation is provided through two levels of indirection. The first level of indirection provides separation from the system-level allocation/deallocation functions. Functions `genalloc` and `genfree` (defined in the Machine Dependent Module) provide the first level of indirection. Another level of indirection is needed to allow for efficient memory usage. This second level of indirection provides efficiency by taking advantage of the fact that data structures of the same size are constantly being requested and freed by CLIPS. If memory is constantly being requested from the system, freed to the system, and then immediately rerequested from the system, a great deal of inefficiency can occur. If memory requested and then freed by CLIPS is maintained by internal CLIPS memory management routines, much of the overhead of constantly requesting, freeing, and rerequesting memory can be avoided.

The Memory Module also contains routines which perform block memory management. These routines provide another level of indirection for memory management if desired. Block memory management routines request large blocks of memory from the system and split these large blocks to provide memory for smaller requests. On certain machines, this can provide increased efficiency if the system-level memory management routines are not very efficient at handling small blocks of memory.

CLIPS memory management of free memory blocks utilizes an array of pointers to memory blocks (the MemoryTable). The array index refers to the memory size being stored in that location. Requests for memory of sizes greater than the size of the MemoryTable are requested from the system. A request with this range would retrieve the block of memory pointed to by the array (if one exists). Returned memory would be added to the linked list of memory already stored in the array. The first four bytes of all memory would be used as a pointer to the next block of memory (hence, the 4-byte memory restriction for requests). Requests of less than four bytes are automatically converted to 4-byte requests.

Functions defined in this module should be used by external functions that wish to utilize memory in conjunction with the CLIPS kernel. The use of functions `genalloc` and `genfree` should be avoided in external functions.

GLOBAL VARIABLES

MemoryTable

PURPOSE: A table containing free memory of various sizes.

C IMPLEMENTATION: Currently implemented as an array of pointers to various sizes of memory. For example, array location 7 would have a pointer to the first free block of memory of size 7. Each memory block uses its first four bytes as a pointer to the next free block of memory; therefore, memory blocks of less than size 4 cannot be stored in the memory table.
TempMemoryPtr

PURPOSE: Provides a global temporary pointer for use with deallocation macros.

TempSize

PURPOSE: Provides a global temporary integer for use with allocation of variable size structure macros.

INTERNAL VARIABLES

BlockInfoSize

PURPOSE: Amount of space needed to store information pertaining to a block of memory. Only defined when block memory management is in use.

BlockMemoryInitialized

PURPOSE: Boolean variable indicating whether block memory management has been initialized. Only defined when block memory management is in use.

ChunkInfoSize

PURPOSE: Amount of space needed to store information pertaining to a chunk of memory that has been allocated for use. Only defined when block memory management is in use.

ConserveMemory

PURPOSE: Boolean flag which indicates whether or not memory should be conserved. If TRUE, then pretty print representations of constructs are not stored.

MemoryAmount

PURPOSE: Contains amount of memory allocated by CLIPS. Does not include overhead associated with maintaining the memory.

MemoryCalls

PURPOSE: Contains total number of outstanding memory requests.
**OutOfMemoryFunction**

**PURPOSE:** A pointer to a function which is to be called when CLIPS cannot satisfy a memory request. This function either exits CLIPS or attempts to free the requested amount of memory.

**TopMemoryBlock**

**PURPOSE:** Pointer to the top block allocated by the block memory manager. Only defined when block memory management is in use.

**GLOBAL FUNCTIONS**

**ActualPoolSize**

**PURPOSE:** Indicates how much memory CLIPS has available in its free pool. On IBM PC DOS machines, the overhead associated with allocation is also included.

**RETURNS:** The number of bytes in the CLIPS free pool of memory (plus overhead on IBM PC DOS machines).

**CopyMemory**

**PURPOSE:** Copies data structures from a source to a destination.

**ARGUMENTS:** The type of structures being copied, the number of structures to copy, a pointer to the destination memory, and a pointer to the source memory.

**C IMPLEMENTATION:** Implemented as a macro. Calls the function `genmemcpy` to copy the memory.

**DefaultOutOfMemoryFunction**

**PURPOSE:** The default function which is called when CLIPS runs out of memory. Prints an “Out of memory” message.

**ARGUMENTS:** The size of the memory block which could not be allocated (this argument is unused).

**RETURNS:** A non-zero value indicating that the memory request cannot be satisfied and that CLIPS should be exited.
### genalloc

**PURPOSE:** Generic memory allocation function which provides a level of indirection.

**ARGUMENTS:** Size of memory requested.

**RETURNS:** A memory block of the appropriate size.

**OTHER NOTES:** `genalloc` uses either `malloc` or `RequestChunk` depending upon whether block memory allocation is being performed. If `genalloc` cannot get the requested memory, it will release all free memory used by CLIPS to the system. It will then try to allocate the memory again, returning whether it succeeds or fails. Note that this function is not called by the CLIPS kernel with the exception of the Memory Module, which provides another level of memory allocation indirection.

### genfree

**PURPOSE:** Generic memory release function which provides a level of indirection.

**ARGUMENTS:** A block of memory and the memory size.

**OTHER NOTES:** `genfree` uses either `free` or `ReturnChunk` depending upon whether block memory allocation is being performed. Note that this function is not called by the CLIPS kernel with the exception of the Memory Module, which provides another level of memory deallocation indirection.

### genlongalloc

**PURPOSE:** Generic memory allocation function which provides a level of indirection.

**ARGUMENTS:** Size of memory requested as a long integer.

**RETURNS:** A memory block of the appropriate size.

**C IMPLEMENTATION:** If the size of an integer is the same as the size of a long integer or if the long integer can be truncated to an integer, then `genalloc` is used to satisfy the request. In addition, special code is included to handle long integer memory requests for the Macintosh and IBM PC computers. If the request cannot be satisfied because the long integer value cannot be truncated to an integer, then CLIPS is exited.
genlongfree

PURPOSE: Generic memory release function which provides a level of indirection.

ARGUMENTS: A block of memory and the memory size as a long integer.

genmemcpy

PURPOSE: Generic memory copy function which provides a level of indirection.

ARGUMENTS: A pointer to a block of memory to be copied, a pointer to a block of memory to store the copied memory, and the amount of memory to be copied.

RETURNS: No meaningful value.

genrealloc

PURPOSE: Generic memory reallocation function which provides a level of indirection.

ARGUMENTS: A block of memory, the size of the memory block, and the new desired size of the memory block.

RETURNS: A memory block of the new size with the contents of the original memory block.

OTHER NOTES: Current implementation is not very sophisticated. The new block is allocated using genalloc, the content of the old block is copied to the new block, and then the old block is freed using genfree.

GetConserveMemory

PURPOSE: Returns the current value of the ConserveMemory flag.

RETURNS: A boolean value.

get_struct

PURPOSE: Allocates memory needed for a structure.

ARGUMENTS: A structure name.

C IMPLEMENTATION: Implemented as a macro. Uses the global variable TempMemoryPtr to provide a temporary pointer.
**get_var_struct**

**PURPOSE:** Allocates memory needed for a structure of varying size.

**ARGUMENTS:** A structure name and the size of the variable length portion of the structure.

**C IMPLEMENTATION:** Implemented as a macro. Uses the global variable `TempMemoryPtr` to provide a temporary pointer.

**gm1**

**PURPOSE:** Allocates a block of memory from the CLIPS maintained pool of free memory. Initializes the contents of the memory to zero.

**ARGUMENTS:** Size of memory required.

**C IMPLEMENTATION:** Searches `MemoryTable` for free memory of the appropriate size. Calls `genalloc` if it cannot find memory of the appropriate size.

**gm2**

**PURPOSE:** Allocates a block of memory from the CLIPS maintained pool of free memory. Does not initialize the contents of the memory.

**ARGUMENTS:** Size of memory required.

**C IMPLEMENTATION:** Searches `MemoryTable` for free memory of the appropriate size. Calls `genalloc` if it cannot find memory of the appropriate size.

**gm3**

**PURPOSE:** Allocates a block of memory from the CLIPS maintained pool of free memory. Does not initialize the contents of the memory.

**ARGUMENTS:** Size of memory required (a long integer).

**C IMPLEMENTATION:** Searches `MemoryTable` for free memory of the appropriate size. Calls `genlongalloc` if it cannot find memory of the appropriate size.

**MemoryRequests**

**PURPOSE:** Returns number of memory requests currently outstanding.
RETURNS: Number of memory requests currently outstanding.

OTHER NOTES: Uses variables incremented and decremented by `genalloc` and `genfree`.

---

### MemoryUsed

PURPOSE: Returns amount of memory currently allocated by CLIPS.

RETURNS: Amount of memory currently used by CLIPS.

OTHER NOTES: Uses variables incremented and decremented by `genalloc` and `genfree`. May not include overhead memory.

---

### PoolSize

PURPOSE: Indicates how much memory CLIPS has available in its free pool.

RETURNS: The number of bytes in the CLIPS free pool of memory.

---

### ReleaseMemory

PURPOSE: Releases a specified amount of free memory maintained by CLIPS back to the system.

ARGUMENTS: A number which indicates when to stop. If the number is -1, all memory will be released. Otherwise, the function will stop when the amount of memory released has exceeded the number. Another argument specifies whether a message is to be printed when CLIPS releases memory.

---

### RequestChunk

PURPOSE: Allocates memory by returning a chunk of memory from a larger block of memory.

ARGUMENTS: Size of memory needed.

C IMPLEMENTATION: Implemented using several functions.

---

### ReturnChunk

PURPOSE: Frees memory allocated using `RequestChunk`.

ARGUMENTS: A pointer to the memory and size of the memory.

C IMPLEMENTATION: Implemented using several functions.
**rm**

**PURPOSE:** Returns a block of memory to the CLIPS maintained pool of free memory.

**ARGUMENTS:** A pointer to a block of memory and a size argument.

**C IMPLEMENTATION:** Adds memory to the appropriate location in the `MemoryTable`. The first four bytes of the memory block are modified to point to the next block of free memory of the same size.

**rm3**

**PURPOSE:** Returns a block of memory to the CLIPS maintained pool of free memory.

**ARGUMENTS:** A pointer to a block of memory and a size argument.

**C IMPLEMENTATION:** Adds memory to the appropriate location in the `MemoryTable`. The first four bytes of the memory block are modified to point to the next block of free memory of the same size. Calls `genlongfree` to return the memory if it cannot be placed in the `MemoryTable`.

**rtn_struct**

**PURPOSE:** Returns memory needed for a structure to the CLIPS maintained pool of free memory.

**ARGUMENTS:** A structure name and a pointer to the structure.

**C IMPLEMENTATION:** Implemented as a macro. Uses the global variable `TempMemoryPtr` for temporary storage.

**rtn_var_struct**

**PURPOSE:** Returns memory needed for a structure of varying size to the CLIPS maintained pool of free memory.

**ARGUMENTS:** A structure name, the size of the variable length portion of the structure, and a pointer to the structure.

**C IMPLEMENTATION:** Implemented as a macro. Uses the global variables `TempMemoryPtr` and `TempSize` for temporary storage.

**SetConserveMemory**

**PURPOSE:** Sets the current value of the `ConserveMemory` flag.
ARGUMENTS: A boolean value (the new value of the flag).
RETURNS: A boolean value (the old value of the flag).

**SetOutOfMemoryFunction**

PURPOSE: Allows the function which is called when CLIPS runs out of memory to be changed.

ARGUMENTS: A pointer to a function which returns an integer and has a single integer argument. The argument to the function is the size of the memory request that could not be satisfied. The return value of the function should be zero if CLIPS should attempt to allocate the memory again or non-zero if CLIPS should not attempt to allocate the memory again (and exit).

RETURNS: A pointer to the previous out of memory function.

**UpdateMemoryRequests**

PURPOSE: Allows the number of memory requests to CLIPS to be updated.

ARGUMENTS: A signed integer value to be added to the number of memory requests currently outstanding.

RETURNS: Updated number of memory requests currently outstanding.

**UpdateMemoryUsed**

PURPOSE: Allows the amount of memory used by CLIPS to be updated.

ARGUMENTS: A signed integer value to be added to the amount of memory currently used by CLIPS.

RETURNS: Updated amount of memory currently used by CLIPS.

**INTERNAL FUNCTIONS**

**AllocateBlock**

PURPOSE: Adds a new block of memory to the list of memory blocks.

ARGUMENTS: Size of new block and a pointer to the last block of memory being managed by the memory manager.
### AllocateChunk

**PURPOSE:** Allocates a chunk of memory for use. Called by `RequestChunk` when it finds a memory chunk of the appropriate size.

**ARGUMENTS:** A pointer to the memory block information record, a pointer to the memory chunk information record, and the size of memory requested.

**RETURNS:** Nothing. Updates information records for future memory management.

### InitializeBlockMemory

**PURPOSE:** Initializes block memory management and allocates the first block.

**ARGUMENTS:** Size of the initial block.
Symbolic data in the form of words and strings must be handled efficiently both in terms of speed and storage management. CLIPS storage management of symbols requires that multiple copies of a symbol be stored in the same location. To accomplish this goal, CLIPS uses a **SymbolTable** to store all occurrences of symbols. For example, the fact (data red green red) would require three entries in the **SymbolTable**: one each for the symbols data, red, and green. The **SymbolTable** also must keep track of symbols that are no longer in use and remove them. To accomplish this, each symbol is given a count to indicate the number of references to the symbol. In the above example (assuming no other previous entries in the **SymbolTable**), symbols data and green would each have a count of 1 while symbol red would have a count of 2. If at any time a symbol has a count of 0, it is no longer necessary to maintain the symbol and it may be removed.

Symbols not expected to remain in the **SymbolTable** are labeled as ephemeral. All symbols initially added to the **SymbolTable** are marked as ephemeral. These symbols have a count of 0 but are not yet removed. The set of all ephemeral symbols is maintained in a list. At certain times, the **EphemeralSymbolList** is traversed to remove unneeded symbols from the **SymbolTable**. Ephemeral symbols that still have a count of 0 are removed from the symbol table, while ephemeral symbols that have a count greater than 0 are left in the **SymbolTable** and their ephemeral status is lost. As an example, consider the following top-level command:

```
CLIPS> (str-cat "red" "blue")
"redblue"
CLIPS>
```

Four symbols are created during execution of this command. The symbols **str-cat**, **red**, and **blue** are added to the **SymbolTable** when the command is parsed, and the symbol **redblue** is added during the execution of the **str-cat** command. Each of these symbols is labeled as ephemeral. After execution of this command, none of the symbols is needed and all can be removed from the **SymbolTable**. Now consider the following command:

```
CLIPS> (assert (data =(str-cat "red" "blue")))
CLIPS>
```

Six symbols are created during execution of this command. The symbols **assert**, **data**, **str-cat**, **red**, and **blue** are added to the **SymbolTable** when the command is parsed, and the symbol **redblue** is added during the evaluation of the **str-cat** function. This command asserts the fact (data redblue) which contains the symbols **data** and **redblue**. The locations in the **SymbolTable** of these two symbols will have their count incremented by one to reflect that another non-ephemeral reference to the symbol is being made. After execution of this command, the symbols **assert**, **str-cat**, **red**, and **blue** could be removed from the **SymbolTable**, whereas the symbols **data** and **redblue** would have to remain.

As stated previously, all symbols are initially marked as ephemeral. This ensures that temporary symbols created during the parsing of commands and the evaluation of functions are easily removed. A symbol can have its count incremented in a variety of ways including the use of a symbol as part of a construct (such as a defrule, deffacts,
or defclass) or the use of a symbol as part of a fact or an instance. As a corollary, the count of a symbol is decremented whenever the corresponding item which refers to that symbol is removed (such as deleting a construct or retracting a fact). The EphemeralSymbolList is periodically checked for symbols that can be removed from the SymbolTable. These periodic checks occur at various times including after the execution of a rule, deffunction, generic function, message-handler, or top-level command.

Because symbols can be created at different evaluation depths (see the Evaluation Module), it is also necessary to store the evaluation depth at which the symbol was created. Ephemeral symbols are not deleted unless they have a count of zero and the ephemeral symbol is being removed at an evaluation depth less than the depth at which the symbol was created.

In addition to symbols, floating point and integer values are also stored in tables. Floating point values are stored in the FloatTable and integer values are stored in the IntegerTable. The operation of these tables is virtually identical to the SymbolTable (with the primary exception being that they are used to store floats and integers rather than strings). The SymbolTable is used to store the values for the CLIPS data types symbol, string, and instance name (i.e. red, "red", and [red] all have the same location in the SymbolTable). The IntegerTable is only used for storing the CLIPS data type integer and the FloatTable is only used for storing the CLIPS data type float. Note that since each symbol, float, or integer data value is represented by a unique pointer value into a table, comparisons of values can be accomplished by comparing these pointer values (although types must also be compared to distinguish between symbols, strings, and instance names).

Symbols can also be linked to other symbols via a relatedSymbol field. In CLIPS 5.1, this field is used only in COOL to conveniently determine the slot name symbol from a slot-accessor message. For example, the get-temperature slot-accessor symbol would be linked to the slot name symbol temperature.

### GLOBAL VARIABLES

<table>
<thead>
<tr>
<th>CLIPSFalseSymbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: A pointer, useful for comparison, to the symbol table entry of the FalseSymbol generated using the AddSymbol function.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CLIPSTrueSymbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: A pointer, useful for comparison, to the symbol table entry of the TrueSymbol generated using the AddSymbol function.</td>
</tr>
</tbody>
</table>
INTERNAL VARIABLES

**EphemeralFloatList**

PURPOSE: A list of pointers to ephemeral floats currently in the FloatTable.

C IMPLEMENTATION: Implemented as a linked list.

**EphemeralIntegerList**

PURPOSE: A list of pointers to ephemeral integers currently in the IntegerTable.

C IMPLEMENTATION: Implemented as a linked list.

**EphemeralSymbolList**

PURPOSE: A list of pointers to ephemeral symbols currently in the SymbolTable.

C IMPLEMENTATION: Implemented as a linked list.

**FalseSymbol**

PURPOSE: The character string that CLIPS uses for the boolean value FALSE. The value of this string is “FALSE”, however, it could be changed to another value such as “WRONG”.

**FloatTable**

PURPOSE: Stores all floats used by CLIPS.

C IMPLEMENTATION: Implemented as an array. Each entry corresponds to a list of float table entries. Collisions are resolved by adding the float entry to list of entries.

**IntegerTable**

PURPOSE: Stores all integers used by CLIPS.

C IMPLEMENTATION: Implemented as an array. Each entry corresponds to a list of integer table entries. Collisions are resolved by adding the integer entry to list of entries.

**SymbolTable**

PURPOSE: Stores all symbols used by CLIPS.
C IMPLEMENTATION: Implemented as an array. Each entry corresponds to a list of symbol table entries. Collisions are resolved by adding the symbol entry to list of entries.

**TrueSymbol**

**PURPOSE:** The character string that CLIPS uses for the boolean value TRUE. The value of this string is “TRUE”, however, it could be changed to another value such as “RIGHT”.

**GLOBAL FUNCTIONS**

**AddDouble**

**PURPOSE:** Adds a double precision floating-pointer number to the FloatTable.

**ARGUMENTS:** A double precision floating point number that is to be added to the FloatTable.

**RETURNS:** The address of the float entry structure for the given number in the FloatTable.

**AddLong**

**PURPOSE:** Adds a long integer to the IntegerTable.

**ARGUMENTS:** A long integer that is to be added to the IntegerTable.

**RETURNS:** The address of the integer entry structure for the given integer in the IntegerTable.

**AddSymbol**

**PURPOSE:** Adds a symbol to the SymbolTable.

**ARGUMENTS:** A string that is to be added to the SymbolTable.

**RETURNS:** The address of the symbol entry structure for the given string in the SymbolTable.

**DecrementFloatCount**

**PURPOSE:** Decrements the count value for a FloatTable entry. Adds the float to the EphemeralFloatList if the count becomes zero.

**ARGUMENTS:** A FloatTable entry.
DecrementIntegerCount

PURPOSE: Decrements the count value for an IntegerTable entry. Adds the integer to the EphemeralIntegerList if the count becomes zero.

ARGUMENTS: An IntegerTable entry.

DecrementSymbolCount

PURPOSE: Decrements the count value for a SymbolTable entry. Adds the symbol to the EphemeralSymbolList if the count becomes zero.

ARGUMENTS: A SymbolTable entry.

FindSymbol

PURPOSE: Determines if a symbol is already in the SymbolTable.

ARGUMENTS: A string that is to be searched for in the SymbolTable.

RETURNS: If the string is contained in the SymbolTable, the address of the symbol entry structure for the given string in the SymbolTable is returned, otherwise NULL is returned.

FindSymbolMatches

PURPOSE: Finds all symbols in the SymbolTable which begin with a specified symbol. This function is used to implement the command completion feature found in some of the CLIPS machine specific interfaces.

ARGUMENTS: A pointer to a string and a pointer to an integer.

RETURNS: Returns a pointer to a list of symbols which begin with the specified sequence of characters. The number of matches is stored in the integer passed as an argument.

GetFloatTable

PURPOSE: Returns a pointer to the FloatTable.

RETURNS: A pointer to the FloatTable.

OTHER NOTES: Normally used by the construct compiler and binary save to gain access to the FloatTable.
**GetIntegerTable**

PURPOSE: Returns a pointer to the **IntegerTable**.

RETURNS: A pointer to the **IntegerTable**.

OTHER NOTES: Normally used by the construct compiler and binary save to gain access to the **IntegerTable**.

---

**GetNextSymbolMatch**

PURPOSE: Finds the next symbol in the **SymbolTable** which begins with a specified symbol. This function is used to implement the command completion feature found in some of the CLIPS machine specific interfaces.

ARGUMENTS: A pointer to a string, the number of characters to use in performing the comparison of strings, and the previous symbol in the symbol table which was checked.

RETURNS: Returns a pointer to the next **SymbolTable** entry which begins with the specified sequence of characters.

---

**GetSymbolTable**

PURPOSE: Returns a pointer to the **SymbolTable**.

RETURNS: A pointer to the **SymbolTable**.

OTHER NOTES: Normally used by the construct compiler and binary save to gain access to the **SymbolTable**.

---

**HashFloat**

PURPOSE: Computes a hash value for a float.

ARGUMENTS: A float and maximum value for the hash value.

RETURNS: An integer hash value which is less than the maximum value.

C IMPLEMENTATION: The float number is converted to a long integer through the use of a union structure to yield a hash value. This value is then divided by the maximum value and the remainder is returned.

---

**HashInteger**

PURPOSE: Computes a hash value for an integer.
ARGUMENTS: An integer and maximum value for the hash value.
RETURNS: An integer hash value which is less than the maximum value.
C IMPLEMENTATION: The integer value is used as the hash value. This value is then divided by the maximum value and the remainder is returned.

**HashSymbol**

PURPOSE: Computes a hash value for a symbol.
ARGUMENTS: A string and maximum value for the hash value.
RETURNS: An integer hash value which is less than the maximum value.
C IMPLEMENTATION: The characters of the string are grouped together to form long integers which are then added together to yield a hash value. This value is then divided by the maximum value and the remainder is returned.

**IncrementFloatCount**

PURPOSE: Increments the count value for a FloatTable entry.
ARGUMENTS: A FloatTable entry.

**IncrementIntegerCount**

PURPOSE: Increments the count value for an IntegerTable entry.
ARGUMENTS: An IntegerTable entry.

**IncrementSymbolCount**

PURPOSE: Increments the count value for a SymbolTable entry.
ARGUMENTS: A SymbolTable entry.

**InitializeAtomTables**

PURPOSE: Initializes the SymbolTable, IntegerTable, and FloatTable. It also initializes the CLIPSTrueSymbol and CLIPSFalseSymbol.
**RefreshBooleanSymbols**

**PURPOSE:** Resets the values of the **CLIPSTrueSymbol** and the **CLIPSFalseSymbol**.

**OTHER NOTES:** Normally called during initialization of a run-time module generated using the constructs-to-c function.

**RemoveEphemeralAtoms**

**PURPOSE:** Causes the removal of all ephemeral symbols, integers, and floats, that still have a count value of zero, from their respective storage tables. This function performs this action by calling the functions **RemoveEphemeralSymbols**, **RemoveEphemeralIntegers**, and **RemoveEphemeralFloats**.

**ReturnSymbolMatches**

**PURPOSE:** Returns a set of symbol matches.

**ARGUMENTS:** A pointer to a list of symbol matches found using the **FindSymbolMatches** function.

**SetFloatTable**

**PURPOSE:** Sets the value of the **FloatTable**.

**ARGUMENTS:** A pointer to a **FloatTable**.

**OTHER NOTES:** Normally used by a run-time module generated using the constructs-to-c function to install the **FloatTable**.

**SetIntegerTable**

**PURPOSE:** Sets value of the **IntegerTable**.

**ARGUMENTS:** A pointer to a **IntegerTable**.

**OTHER NOTES:** Normally used by a run-time module generated using the constructs-to-c function to install the **IntegerTable**.

**SetSymbolTable**

**PURPOSE:** Sets value of the **SymbolTable**.

**ARGUMENTS:** A pointer to a **SymbolTable**.
OTHER NOTES: Normally used by a run-time module generated using the constructs-to-c function to install the SymbolTable.

INTERNAL FUNCTIONS

AddEphemeralFloat

PURPOSE: Adds a float to the EphemeralFloatList.
ARGUMENTS: A FloatTable entry.
OTHER NOTES: Typically called when a float is added to the FloatTable or when a float's count value reaches zero.

AddEphemeralInteger

PURPOSE: Adds an integer to the EphemeralIntegerList.
ARGUMENTS: An IntegerTable entry.
OTHER NOTES: Typically called when an integer is added to the IntegerTable or when an integer's count value reaches zero.

AddEphemeralSymbol

PURPOSE: Adds a symbol to the EphemeralSymbolList.
ARGUMENTS: A SymbolTable entry.
OTHER NOTES: Typically called when a symbol is added to the SymbolTable or when a symbol's count value reaches zero.

RemoveEphemeralFloats

PURPOSE: Removes all ephemeral floats from the FloatTable that still have a count value of zero and were created at a evaluation depth greater than the current evaluation depth. Uses the EphemeralFloatList to determine which floats to check. Floats that have a count greater than zero are removed from the EphemeralFloatList.

RemoveEphemeralIntegers

PURPOSE: Removes all ephemeral integers from the IntegerTable that still have a count value of zero and were created at a evaluation depth greater than the current evaluation depth.
Uses the **Ephemeral IntegerList** to determine which integers to check. Integers that have a count greater than zero are removed from the **EphemeralIntegerList**.

### RemoveEphemeralSymbols

**PURPOSE:** Removes all ephemeral symbols from the **SymbolTable** that still have a count value of zero and were created at a evaluation depth greater than the current evaluation depth. Uses the **EphemeralSymbolList** to determine which symbols to check. Symbols that have a count greater than zero are removed from the **EphemeralSymbolList**.

### RemoveFloat

**PURPOSE:** Removes a float from the **FloatTable**.

**ARGUMENTS:** A **FloatTable** entry.

### RemoveInteger

**PURPOSE:** Removes an integer from the **IntegerTable**.

**ARGUMENTS:** An **IntegerTable** entry.

### RemoveSymbol

**PURPOSE:** Removes a symbol from the **SymbolTable**.

**ARGUMENTS:** A **SymbolTable** entry.
The Router Module (router.c) provides a level of indirection between low-level I/O implementations and high-level requests for I/O. All high-level requests for I/O are directed to logical names. The logical names are then associated with specific I/O implementations. Changing the CLIPS interface using this technique is now made very easy. To change the interface from a command line interface to a windowed interface only requires reassociating the appropriate logical names with I/O implementations for windows. High-level requests do not need to be changed. More details of the I/O Router mechanism can be found in Section 7 of the Advanced Programming Guide.

GLOBAL VARIABLES

**CLIPSInputCount**

**PURPOSE:** Integer used to keep track of the number of characters currently entered while CLIPS is accepting input. Used by some of the machine specific interfaces to prevent backing over output (such as the CLIPS prompt) when input is being deleted.

**WCLIPS**

**PURPOSE:** Global variable which can be used to refer to the `wclips` logical name.

**WDIALOG**

**PURPOSE:** Global variable which can be used to refer to the `wdialog` logical name.

**WDISPLAY**

**PURPOSE:** Global variable which can be used to refer to the `wdisplay` logical name.

**WERRO**

**PURPOSE:** Global variable which can be used to refer to the `werror` logical name.

**WTRACE**

**PURPOSE:** Global variable which can be used to refer to the `wtrace` logical name.
INTERNAL VARIABLES

Abort

PURPOSE: Boolean flag which indicates if the ExitCLIPS call should be aborted without exiting CLIPS.

FastLoadFilePtr

PURPOSE: Variable which indicates whether I/O router system is to be bypassed and input performed directly from a file.

C IMPLEMENTATION: If FastLoadFilePtr is NULL, regular I/O router procedure is used. If FastLoadFilePtr is not NULL, it is the file pointer to which I/O should be performed.

FastSaveFilePtr

PURPOSE: Variable which indicates whether I/O router system is to be bypassed and output performed directly to a file.

C IMPLEMENTATION: If FastSaveFilePtr is NULL, regular I/O router procedure is used. If FastSaveFilePtr is not NULL, it is the file pointer to which I/O should be performed.

ListOfFileRouters

PURPOSE: List of all defined file routers. File routers provide a mechanism for reading and writing to files. File routers are created using the open command.

ListOfRouters

PURPOSE: List of all defined I/O routers.

C IMPLEMENTATION: Router structure has information on router name, priority, boolean active flag, query function, print function, exit function, get character function, unget character function, and a pointer to the next router. The routers are linked in order of priority.

ListOfStringRouters

PURPOSE: List of all defined string routers. String routers provide a mechanism for reading input from a string or writing output to a string.
GLOBAL FUNCTIONS

AbortExit

PURPOSE: Sets the value of the Abort flag to TRUE.

ActivateRouter

PURPOSE: Activates a specified router.
ARGUMENTS: Name of router.

AddRouter

PURPOSE: Adds an I/O router to the ListOfRouters. The router is placed before routers with a lower priority and after routers with a higher priority.
ARGUMENTS: Router name, priority, boolean active flag, query function, print function, exit function, get character function, unget character function.
OTHER NOTES: Routers are active when created.

CloseAllFiles

PURPOSE: Closes all opened files.

CloseFile

PURPOSE: Closes a file.
ARGUMENTS: The logical name associated with the file when opened with OpenFile.

CloseStringDestination

PURPOSE: Closes a string output destination.
ARGUMENTS: Name of string router used when created with OpenStringDestination.

CloseStringSource

PURPOSE: Closes a string input source.
ARGUMENTS: Name of string router used when created with OpenStringSource.
### DeactivateRouter

**PURPOSE:** Deactivates a specified router.

**ARGUMENTS:** Name of router.

### DeleteRouter

**PURPOSE:** Removes an I/O router from the *ListOfRouters*.

**ARGUMENTS:** Name of I/O router.

**RETURNS:** Boolean value. TRUE if the router was successfully deleted, otherwise FALSE.

### ExitCLIPS

**PURPOSE:** High-level CLIPS exit routine. Calls all router exit functions before calling genexit function.

**ARGUMENTS:** Exit number.

### FindFile

**PURPOSE:** Determines if a file which the specified logical name has been opened.

**ARGUMENTS:** A logical name.

**RETURNS:** Boolean value. TRUE if a file with the specified logical name has been opened, otherwise FALSE.

### FindFptr

**PURPOSE:** Returns a pointer to an opened file.

**ARGUMENTS:** A logical name.

**RETURNS:** Boolean value. A pointer to the specified file, if found, otherwise NULL.

### GetcCLIPS

**PURPOSE:** High-level request function to get a character.

**ARGUMENTS:** Logical name from which character is requested.

**RETURNS:** A character.
OTHER NOTES: Routine must check for `FastLoadFilePtr` and `FastSaveFilePtr`.

### GetFastLoad

**PURPOSE:** Returns the value of the variable `FastLoadFilePtr`.

### GetFastSave

**PURPOSE:** Returns the value of the variable `FastSaveFilePtr`.

### InitializeDefaultRouters

**PURPOSE:** Initializes the standard I/O routers used by CLIPS (file and string).

### OpenFile

**PURPOSE:** Opens a file for input or output by creating a file router.

**ARGUMENTS:** The name of the file, the mode in which the file is to be opened (read, write, etc.), and the logical name to be associated with the file.

### OpenStringDestination

**PURPOSE:** Allows a string to be used as an output destination by creating a string router.

**ARGUMENTS:** Name to be associated with the string router, the string to which output is sent, and the maximum number of characters that can be sent to the string.

**RETURNS:** Boolean value. TRUE if the string router was successfully created, otherwise FALSE.

### OpenStringSource

**PURPOSE:** Allows a string to be used as a source of input by creating a string router.

**ARGUMENTS:** Name to be associated with the string router, the string from which input is read, and the starting location within the string.

**RETURNS:** Boolean value. TRUE if the string router was successfully created, otherwise FALSE.
**OpenTextSource**

**PURPOSE:** Allows a string to be used as a source of input by creating a string router. Since this function allows the maximum number of characters which can be read from the string to be specified, it is useful for reading from strings which are not NULL terminated and for reading from a substring of a string.

**ARGUMENTS:** Name to be associated with the string router, the string from which input is read, the starting location within the string, and the maximum number of characters which can be read from the string.

**RETURNS:** Boolean value. TRUE if the string router was successfully created, otherwise FALSE.

**PrintCLIPS**

**PURPOSE:** High-level request function to print a string.

**ARGUMENTS:** A string to print and the logical name to which the string is to be printed.

**OTHER NOTES:** Routine must check for `FastLoadFilePointer` and `FastSaveFilePointer`.

**QueryRouters**

**PURPOSE:** Determines if any router recognizes a logical name.

**ARGUMENTS:** Logical name.

**RETURNS:** Boolean value. TRUE if the logical name is recognized by any router, otherwise FALSE.

**SetFastLoad**

**PURPOSE:** Sets value of the variable `FastLoadFilePtr`.

**ARGUMENTS:** Value to which `FastLoadFilePtr` is to be set.

**SetFastSave**

**PURPOSE:** Sets value of the variable `FastSaveFilePtr`.

**ARGUMENTS:** Value to which `FastSaveFilePtr` is to be set.
**UngetcCLIPS**

**PURPOSE:** High-level request function to unget a character.

**ARGUMENTS:** Logical name to which character is ungoten and the character to unget.

**OTHER NOTES:** Routine must check for `FastLoadFilePtr` and `FastSaveFilePtr`.

---

**UnrecognizedRouterMessage**

**PURPOSE:** A generic error message which can be printed when a logical name is not recognized by any routers.

**ARGUMENTS:** The logical name which was unrecognized.

---

**INTERNAL FUNCTIONS**

---

**CreateReadStringSource**

**PURPOSE:** Drive routine for creating a string router for a string input source.

**ARGUMENTS:** Name to be associated with the string router, the string from which input is read, the starting location within the string, and the maximum number of characters which can be read from the string.

**RETURNS:** Boolean value. TRUE if the string router was successfully created, otherwise FALSE.

---

**File Router Functions**

**PURPOSE:** Set of functions needed to handle file routers. Note that this is not a single function but actually a series of functions.

---

**QueryRouter**

**PURPOSE:** Determines if a specific router recognizes a logical name.

**ARGUMENTS:** Logical name and an I/O router.

**RETURNS:** Boolean value. TRUE if the logical name is recognized by the router, otherwise FALSE.
String Router Functions

PURPOSE: Set of functions needed to handle string routers. Note that this is not a single function but actually a series of functions.
Scanner Module

The Scanner Module (scanner.c) "scans" input sources for tokens recognizable by CLIPS. The scanner receives input from logical names as described in the Router Module. The scanner returns token information in a data structure with several fields. One field indicates the type of token. For example, the token 783 would have type INTEGER, the token ( would have type LEFT_PARENTHESIS, and the token "cat" would have type STRING. Another field in the token structure supplies the data value for tokens which have a data value. In the example above, "cat" would have a data value of "cat" (which would be a pointer to the symbol entry for "cat" in the SymbolTable). Note that the symbol cat would have the same data value as the string "cat". In addition, tokens also have a printed representation. The token ?x, for example, would have token type VARIABLE, data type "x", and printed representation "?x".

CLIPS produces a formatted representation for every parsed command or construct. Since this formatting process is closely linked with the scanner, the routines for creating this "pretty print" representation are included in the Scanner Module and directly called by the scanner routines. Every token that is read using the Scanner Module is placed in the PrettyPrintBuffer unless the buffer has been disabled. The buffer is normally disabled during execution of a knowledge base (it is not normally desirable to format input read from a file).

GLOBAL VARIABLES

<table>
<thead>
<tr>
<th>IgnoreCompletionErrors</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Boolean flag which indicates whether an error should be signalled when a string is being scanned and an end-of-file is encountered.</td>
</tr>
</tbody>
</table>

INTERNAL VARIABLES

<table>
<thead>
<tr>
<th>GlobalMax</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: The maximum number of characters which can be stored in GlobalString.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GlobalPos</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: The current number of characters stored in GlobalString.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GlobalString</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Buffer to store string data values for tokens.</td>
</tr>
</tbody>
</table>
**IndentationDepth**

**PURPOSE:** Used by the pretty print functions to determine how many spaces to indent when an indentation command is given.

**PPBufferMax**

**PURPOSE:** The maximum number of characters which can be stored in **PrettyPrintBuffer**.

**PPBufferPos**

**PURPOSE:** The current number of characters stored in **PrettyPrintBuffer**.

**PrettyPrintBuffer**

**PURPOSE:** Buffer to maintain a “pretty” representation of the current command or rule being parsed. Also requires several variables to keep track of current position in buffer.

**PPBackupOnce**

**PURPOSE:** The position to which to backup in the **PrettyPrintBuffer** the first time that **PPBackup** is called.

**PPBackupTwice**

**PURPOSE:** The position to which to backup in the **PrettyPrintBuffer** the second time that **PPBackup** is called.

**PPBufferStatus**

**PURPOSE:** Boolean flag which indicates whether parsed tokens should be stored in the **PrettyPrintBuffer**.

**GLOBAL FUNCTIONS**

**CopyPPBuffer**

**PURPOSE:** Makes a copy of the **PrettyPrintBuffer**.

**RETURNS:** A string copy of the **PrettyPrintBuffer**.

**CopyToken**

**PURPOSE:** Copies values of one token to another token.
ARGUMENTS: Source token and target token.

RETURNS: Nothing. Values of the target token will be set to values of the source token.

**DecrementIndentDepth**

PURPOSE: Decrements `IndentationDepth` for pretty printing.

ARGUMENTS: Value by which `IndentationDepth` is to be decremented.

**DestroyPPBuffer**

PURPOSE: Resets the state of the `PrettyPrintBuffer` to contain nothing and returns the string associated with the pretty print representation to the pool of free memory.

**FlushPPBuffer**

PURPOSE: Resets state of the `PrettyPrintBuffer` to contain nothing.

**GetPPBuffer**

PURPOSE: Returns a pointer to the `PrettyPrintBuffer`.

RETURNS: A pointer to the `PrettyPrintBuffer`.

**GetPPBufferStatus**

PURPOSE: Returns the value of the `PPBufferStatus` flag.

RETURNS: Boolean value.

**GetToken**

PURPOSE: Reads next token from the input stream.

ARGUMENTS: Logical name from which input is read and a pointer to a token structure in which to store the scanned token.

RETURNS: Nothing. The pointer to the token data structure passed as an argument is set to contain the type of token (e.g., symbol, string, integer, etc.), the data value for the token (i.e., a symbol table location if it is a symbol or string, an integer table location if it is an integer), and the pretty print representation.
IncrementIndentDepth

PURPOSE: Increments IndentationDepth for pretty printing.

ARGUMENTS: Value by which IndentationDepth is to be incremented.

PPBackup

PURPOSE: Backs up past last appended string to the PrettyPrintBuffer.

OTHER NOTES: Should only have to be capable of backing up over last two appended strings.

PPCRAAndIndent

PURPOSE: Prints a carriage return (CR) followed by a number of spaces equal to the IndentationDepth of the PrettyPrintBuffer.

SavePPBuffer

PURPOSE: Appends a string to the end of the PrettyPrintBuffer.

ARGUMENTS: String to append to buffer.

SetIndentDepth

PURPOSE: Sets IndentationDepth for pretty printing.

ARGUMENTS: Value to which IndentationDepth is to be set.

SetPPBufferStatus

PURPOSE: Sets PPBufferStatus on or off.

ARGUMENTS: Boolean value. TRUE if PrettyPrintBuffer is to be turned on; FALSE if PrettyPrintBuffer is to be turned off.

OTHER NOTES: PPBufferStatus should be on during rule or command parse and off during rule execution.

StringPrintForm

PURPOSE: Generates printed representation of a string. Replaces / with // and " with ".

ARGUMENTS: A string.

RETURNS: Printed representation of the string.
INTERNAL FUNCTIONS

**AppendStrings**

**PURPOSE:** Appends two strings together.

**ARGUMENTS:** Two pointers to strings.

**RETURNS:** A pointer to a string created by appending the two strings passed as arguments. The string is added to the `SymbolTable` so it is not necessary to deallocate the string returned.

**ScanNumber**

**PURPOSE:** Parses a number.

**ARGUMENTS:** Logical name from which input is read and a token data structure to store the parsed value.

**RETURNS:** The parsed data value in the token structure passed as an argument. The type of the token will either be an integer (in which case the value in the token will be an `IntegerTable` entry), a float (in which case the value in the token will be a `FloatTable` entry), or a symbol otherwise (in which case the value in the token will be an `SymbolTable` entry). The pretty print representation of the data value will also be stored in the token.

**OTHER NOTES:** See the *Basic Programming Guide* for a detailed explanation of the integer and float data types. Note that any data value that first appears to be a number, but does not satisfy the requirements of a number is treated as a symbol (e.g. 37-A).

**ScanString**

**PURPOSE:** Parses a string.

**ARGUMENTS:** Logical name from which input is read.

**RETURNS:** `SymbolTable` entry for the string.

**OTHER NOTES:** See the *Basic Programming Guide* for a detailed explanation of the string data type.

**ScanSymbol**

**PURPOSE:** Parses a symbol.
ARGUMENTS: Logical name from which input is read, the number of characters in the symbol that have already been placed in the StringBuffer, and integer value for storing the symbol's type (since a symbol may actually be an instance name).

RETURNS: SymbolTable entry for the symbol.

OTHER NOTES: See the Basic Programming Guide for a detailed explanation of the symbol data type.
Expression Module

The standard format used by CLIPS for expressions is very similar to a LISP format. In general, expressions follow the format

(function-name arg1 arg2 ... argn)

where each argument may be an expression, a typeable primitive data type (symbol, string, integer, float, or instance name, but not external address or instance), or a variable (either local or global). The function name refers either to a system or user defined function, a deffunction, or a generic function. All of the following would be valid CLIPS expressions:

(facts)
(+ (* 3 (- ?x 3)) 6)
(str-cat "red" "blue")

The Expression Module (expressn.c) contains routines which parse expressions into a format which, in most cases, is suitable for evaluation by the Evaluation Module (evaluatn.c). It also checks that the first symbol found in a function call is a function name. The parsing of constructs (such as defrule and deffacts) is handled by the Constructs Module (constrct.c). In addition, the parsing of certain CLIPS expressions which do not conform to the standard expression format are handled by the Special Forms Module (spclform.c).

The data structure used to store each component of an expression consists of a type field (such as SYMBOL or INTEGER), a value field (such as a pointer to a SymbolTable entry), a pointer to an argument list (for functions), and a pointer to the next argument in the argument list. For example, the following expression

(+ (* 3 (- 8.3 2) 11) 6.5)

would be represented as shown following (with down pointing arrows representing the argument list pointers and right pointing arrows representing the next argument pointer).
GLOBAL VARIABLES

None.

INTERNAL VARIABLES

<table>
<thead>
<tr>
<th>FunctionHashTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
<tr>
<td>C IMPLEMENTATION:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ListOfFunctions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>

GLOBAL FUNCTIONS

<table>
<thead>
<tr>
<th>AddFunctionParser</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
<tr>
<td>ARGUMENTS:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AddHashFunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
<tr>
<td>ARGUMENTS:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ArgumentParse</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>
ARGUMENTS: Logical name from which input is read, and a pointer to an integer in which an error code is returned.

RETURNS: A pointer to an expression representing the next argument in the function call. Note that this value may be null, indicating that no further arguments exist. The error status is passed back through the pointer to an integer passed as a parameter.

**CollectArguments**

PURPOSE: Parses and groups together all of the arguments for a function call expression by repeatedly calling `ArgumentParse`.

ARGUMENTS: Logical name from which input is read a pointer to the function call expression to which the arguments are to be attached.

RETURNS: The pointer to the function call expression with its arguments attached. If an error occurs, the function call expression is returned to the pool of free memory and NULL is returned.

**ConstantExpression**

PURPOSE: Identifies expressions that are constants.

ARGUMENTS: An expression.

RETURNS: Returns TRUE if the expression is a constant (symbol, string, integer, float, instance, or instance name), otherwise FALSE is returned.

**CopyExpression**

PURPOSE: Copies an expression.

ARGUMENTS: Expression to be copied.

RETURNS: A copy of the expression.

**CountArguments**

PURPOSE: Returns the number of arguments associated with an expression (i.e. how many arguments a function call has).

ARGUMENTS: An expression.
RETURNS: Returns an integer value representing the number of arguments found.

**ExpressionContainsVariables**

PURPOSE: Determines if an expression contains any variables.

ARGUMENTS: An expression and a boolean flag indicating whether global variables should be considered as variables.

RETURNS: Returns TRUE if the expression contains any variables, otherwise FALSE is returned.

**ExpressionDeinstall**

PURPOSE: Decrements count values for generic functions, deffunctions, and constant values (such as symbols) for all such occurrences found in an expression.

ARGUMENTS: An expression.

**ExpressionInstall**

PURPOSE: Increments count values for generic functions, deffunctions, and constant values (such as symbols) for all such occurrences found in an expression.

ARGUMENTS: An expression.

**ExpressionSize**

PURPOSE: Returns the total number of nodes contained in an expression.

ARGUMENTS: An expression (packed or unpacked).

RETURNS: Returns an integer value representing the total number of nodes in the expression.

**Function0Parse**

PURPOSE: Parses a function call. Assumes that none of the functions has been parsed yet.

ARGUMENTS: Logical name from which input is read.

RETURNS: A pointer to an expression. Returns null if an error occurs.
**Function1Parse**

**PURPOSE:** Parses a function call. Assumes that the opening left parenthesis of the function has already been parsed.

**ARGUMENTS:** Logical name from which input is read.

**RETURNS:** A pointer to an expression. Returns null if an error occurs.

**Function2Parse**

**PURPOSE:** Parses a function call. This routine is able to distinguish between system and user defined functions, deffunctions, and generic functions. If the routine has a specialized parsing routine, then that routine will be called by this routine in place of the default argument parsing routine. This routine assumes that the opening left parenthesis and the name of the function have already been parsed.

**ARGUMENTS:** Logical name from which input is read and name of the function to be parsed.

**RETURNS:** A pointer to an expression. Returns null if an error occurs.

**FindFunction**

**PURPOSE:** Determines if a function has been defined using the function `DefineFunction`.

**ARGUMENTS:** A function name.

**RETURNS:** A pointer to the function entry if it exists, otherwise NULL.

**GetFunctionList**

**PURPOSE:** Returns the `ListOfFunctions`.

**IdenticalExpression**

**PURPOSE:** Determines if two expressions are identical.

**ARGUMENTS:** Two expressions.

**RETURNS:** Returns TRUE if the expressions are identical, otherwise FALSE is returned.
InstallFunctionList

PURPOSE: Sets the ListOfFunctions and adds all the function entries to the FunctionHashTable.

ARGUMENTS: A linked list of function entries.

OTHER NOTES: Normally used by a run-time module generated using the constructs-to-c function to install the list of functions used by the module.

ListToPacked

PURPOSE: Copies a list of expressions to an array.

ARGUMENTS: A pointer to the expression list to be copied, a pointer to the array to which the expression is to be copied, and an integer index indicating the starting point in the array at which the copying should begin.

RETURNS: The last array index into which the expression was copied.

PackExpression

PURPOSE: Copies an expression (created using multiple memory requests) into an array (created using a single memory request) while maintaining all appropriate links in the expression. A packed expression requires less total memory because it reduces the overhead required for multiple memory allocations.

ARGUMENTS: The expression to be packed.

RETURNS: A copy of the expression packed into an array.

ParseAtomOrExpression

PURPOSE: Parses an expression which may be a function call, atomic value (string, symbol, etc.), or variable (local or global).

ARGUMENTS: Logical name from which input is read.

RETURNS: A pointer to an expression. Returns NULL if an error occurs.

ParseConstantArguments

PURPOSE: Creates an argument list from a series of constants found in a string.
ARGUMENTS: A string and a pointer to an integer.

RETURNS: A pointer to an expression. The integer passed as a parameter is set to TRUE if an error occurs.

PrintExpression

PURPOSE: Prints an expression.

ARGUMENTS: An expression and the logical name to which output is to be sent.

RemoveFunctionParser

PURPOSE: Removes a specialized expression parsing function (if it exists) from the function entry for a function.

ARGUMENTS: Name of function whose parsing function is to be removed.

ReturnExpression

PURPOSE: Returns an expression to the memory manager.

ARGUMENTS: An expression.

OTHER NOTES: If expression was installed using InstallExpression it should be deinstalled using DeinstallExpression before this function is called.

ReturnPackedExpression

PURPOSE: Returns a packed expression created using PackExpression to the memory manager.

ARGUMENTS: A packed expression.

OTHER NOTES: If expression was installed using InstallExpression it should be deinstalled using DeinstallExpression before this function is called.

SetFunctionList

PURPOSE: Sets the ListOfFunctions.

ARGUMENTS: A linked list of function entries.
INTERNAL FUNCTIONS

**InitializeFunctionHashTable**

PURPOSE: Initializes the FunctionHashTable.
Some CLIPS expressions do not conform to the standard expression format. An example of this type of expression is the assert:

(assert (data 35))

The subexpression found within the assert (data 35) is not a function call to be evaluated but, rather, a piece of data for the assert function. Special parsing is required to allow this format for the assert function. Many other functions such as if, while, bind, and retract either transform the expression in some special way or perform additional syntax checking on the expression format. These functions all require special parsing.

Specialized parsing functions are responsible for constructing an appropriate expression representation, as well as for making the appropriate calls to the pretty print routines to format the expression correctly for output.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

ListOfParsedBindNames

PURPOSE: Contains the list of variables encountered by parsing the bind function.

GLOBAL FUNCTIONS

ClearParsedBindNames

PURPOSE: Clears the ListOfParsedBindNames returning all structures to the pool of free memory.

GetParsedBindNames

PURPOSE: Returns the ListOfParsedBindNames.

InitializeSpecialForms

PURPOSE: Initializes specialized parsing functions for assert, bind, if, while, and retract. Also initializes several parsing functions for some math and predicate functions which provide additional error checking for the arguments of these functions.
### ParsedBindNamesEmpty

**PURPOSE:** Indicates if any bind names have been parsed.

**RETURNS:** Returns TRUE if the `ListOfParsedBindNames` is NULL, otherwise FALSE.

### SearchParsedBindNames

**PURPOSE:** Searches the `ListOfParsedBindNames` for a particular variable name.

**ARGUMENTS:** A variable name.

**RETURNS:** Returns TRUE if the variable was found, otherwise FALSE.

### SetParsedBindNames

**PURPOSE:** Sets the value of the `ListOfParsedBindNames`.

**ARGUMENTS:** A new list of parsed bind names.

### INTERNAL FUNCTIONS

#### AssertParse

**PURPOSE:** Handles special parsing of assert expression.

**ARGUMENTS:** Logical name from which input is read, and a pointer to the expression function call.

**RETURNS:** Expression representing the assert function (or NULL if an error occurs).

#### AddBindName

**PURPOSE:** Adds a variable name to the `ListOfParsedBindNames`.

**ARGUMENTS:** Name of the variable.

#### BindParse

**PURPOSE:** Handles special parsing of bind expression.

**ARGUMENTS:** Logical name from which input is read, and a pointer to the expression function call.
RETURNS: Expression representing the bind function (or NULL if an error occurs).

**CheckArgListParse**

PURPOSE: Handles parsing for functions which require a specified number of arguments of either numeric or non-numeric values.

ARGUMENTS: Logical name from which input is read, a pointer to the expression function call, an integer representing the restriction on the arguments (EXACTLY, AT_LEAST, NO_MORE_THAN, etc.), the number of arguments to which the restriction applies, and a boolean value indicating whether the arguments must be numeric.

RETURNS: Expression representing the parsed function (or NULL if an error occurs).

**IfParse**

PURPOSE: Handles special parsing of if expression.

ARGUMENTS: Logical name from which input is read, and a pointer to the expression function call.

RETURNS: Expression representing the if function (or NULL if an error occurs).

**MultiArgNumericParse**

PURPOSE: Handles parsing for functions which require at least two numeric arguments. Currently used by the following functions: +, *, -, /, <=, >=, <, >, =, <>, **min**, and **max**.

ARGUMENTS: Logical name from which input is read, and a pointer to the expression function call.

RETURNS: Expression representing the parsed function (or NULL if an error occurs).

**MultiArgParse**

PURPOSE: Handles parsing for functions which require at least two arguments. Currently used by the following functions: **and** and **or**.

ARGUMENTS: Logical name from which input is read, and a pointer to the expression function call.
RETURNS: Expression representing the parsed function (or NULL if an error occurs).

**NotParse**

PURPOSE: Handles parsing for functions which require exactly one argument. Currently used by the *not* function.

ARGUMENTS: Logical name from which input is read, and a pointer to the expression function call.

RETURNS: Expression representing the parsed function (or NULL if an error occurs).

**RetractParse**

PURPOSE: Handles special parsing of retract expression.

ARGUMENTS: Logical name from which input is read, and a pointer to the expression function call.

RETURNS: Expression representing the retract function (or NULL if an error occurs).

**WhileParse**

PURPOSE: Handles special parsing of while expression.

ARGUMENTS: Logical name from which input is read, and a pointer to the expression function call.

RETURNS: Expression representing the while function (or NULL if an error occurs).
Parser Utility Module

The Parser Utility Module (parsutil.c) provides a number of function which perform various parsing tasks.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

None.

GLOBAL FUNCTIONS

BuildRHSAssert

PURPOSE: Parses one or more RHS pattern and creates an assert command from the patterns.

ARGUMENTS: A pointer to an assert function call expression (which will be converted to a prog if more than one pattern is to be asserted), logical name from which input is read, a boolean flag indicating if opening right parenthesis of the first RHS pattern has already been parsed, and a pointer to a boolean flag which indicates if a parsing error occurred.

RETURNS: A pointer to an expression (NULL if an error was encountered). The parsing error flag is always set to either TRUE or FALSE by this routine.

CompactActions

PURPOSE: Converts a prog function call expression to a simpler format if it contains less than two arguments. A prog with no arguments if converted to an expression containing the symbol FALSE. A prog with a single argument is converted to an expression containing the single argument.

ARGUMENTS: A pointer to an expression.

RETURNS: A pointer to an expression.

GetAssertArgument

PURPOSE: Parses a single argument for use within an assert command (e.g. a single symbol or variable).
ARGUMENTS: Logical name from which input is read, a pointer to a token structure in which scanned tokens are placed, a pointer to a boolean flag which indicates whether a multifield value was parsed, a pointer to a boolean flag which indicates if a parsing error occurred, the type of token which indicates that no more assert arguments are available (e.g. a right parenthesis), a boolean flag indicating if only constants are allowed to be parsed, and a boolean flag indicating whether an error message should be printed by the calling function when an error is detected by this function.

RETURNS: A pointer to an expression. The multifield flag and error flag are set to TRUE if a multifield or error is encountered while parsing. The print error message flag is always set to either TRUE or FALSE by this routine.

**GetConstructNameAndComment**

PURPOSE: Parses the name and comment fields of a construct. If the construct is being redefined, then the current definition of the construct is deleted. If compilations are being watched then this function will print out an informational message, otherwise a single character is printed to indicate a new construct is being defined.

ARGUMENTS: Logical name from which input is read, a pointer to a token structure in which scanned tokens are placed, the name of the construct type being parsed (e.g. defrule), a pointer to a function which will delete the construct in case the parsed construct is being redefined, the character symbol which is printed to indicate a construct is being defined (e.g. "*" for defrule), and a boolean flag indicating if a carriage return should be printed after the long informational message when compilations are being watched.

RETURNS: The name of the construct being parsed.

**GetRHSPattern**

PURPOSE: Parses the type of pattern typically encountered on the RHS of a rule for functions such as `assert` and `modify`, but can also be found in constructs such as `deffacts`. A RHS pattern consists of a left parenthesis, followed by one or more primitive data types or variables, followed by a right parenthesis. The fields in the RHS pattern may also be specified using a deftemplate format.

ARGUMENTS: Logical name from which input is read, a pointer to a token structure in which scanned tokens are placed, a pointer to a
boolean flag which indicates whether a multifield value was parsed, a pointer to a boolean flag which indicates if a parsing error occurred, a boolean flag indicating if only constants are allowed to be parsed, a boolean flag indicating if opening right parenthesis of the RHS pattern has already been parsed, and the type of token which indicates the end of the RHS pattern (e.g. a right parenthesis).

RETURNS: A pointer to an expression. The multifield flag and error flag are set to TRUE if a multifield or error is encountered while parsing.

OTHER NOTES: Primarily uses the function `GetAssertArgument` to parse an ordered fact and the function `ParseAssertTemplate` to parse a deftemplate fact.

GroupActions

PURPOSE: Parses a series of actions and groups them together in a `progn` command.

ARGUMENTS: Logical name from which input is read, a pointer to a token structure in which scanned tokens are placed, a boolean flag indicating if first token of the group of actions has already been parsed, and the string representation of the type of token which indicates the end of the group of actions (in addition to a right parenthesis).

RETURNS: A pointer to an expression (NULL if an error was encountered).

ReadUntilClosingParen

PURPOSE: Scans tokens until a matching closing right parenthesis is found. This function assumes that an opening left parenthesis has already been parsed before the function was called and verifies that each left parenthesis encountered has a matching right parenthesis.

ARGUMENTS: Logical name from which input is read and a pointer to a token structure in which scanned tokens are placed.

RETURNS: Boolean value. TRUE if the closing right parenthesis was found, otherwise FALSE.

INTERNAL FUNCTIONS

None.
The Evaluation Module (evaluatn.c) provides a set of functions for evaluating expressions. In addition, functions for defining functions and accessing the argument values of expressions are provided.

In versions of CLIPS previous to version 5.0, garbage collection was simplified by that fact that it could be performed on rule firing boundaries. Symbols and other data structures created by the evaluation of expressions could be checked at the end of each rule firing to determine if they could be garbage collected. Version 5.0 of CLIPS, however, introduced object-oriented and procedural programming paradigms. It is now possible to have a CLIPS program which contains no rules at all. Thus, it is no longer sufficient to perform garbage collection only on rule boundaries. Garbage collection of symbols and other ephemeral data structures can now occur at the completion of each rule, deffunction, generic function, or message-handler that is executed.

Because rule firings, function calls, and message passing can be nested many levels deep, it is necessary to associate an “evaluation depth” with each ephemeral data structure that is created. This evaluation depth indicates the levels of unnesting that must occur before a particular data structure can be garbage collected. For example, if function foo calls function bar which in turn calls function yak, then data structures created through the evaluation of expressions in function foo would have an evaluation depth of 1. Similarly, expression evaluation results in function bar would have an evaluation depth of 2 and results from function yak would have an evaluation depth of 3. Ephemeral data structures created at a depth of 3 could be garbage collected upon return to either function foo or bar. Similarly, data structures created at a depth of 2 could be garbage collected upon return to function foo and the data structures created by foo could be garbage collected once foo was exited.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

| **BindList** |
| PURPOSE: A linked list of the local variables that are dynamically allocated by the bind command for a given evaluation depth. Any routine which increments the CurrentEvaluationDepth value must store the old value of the BindList and restore this value when the CurrentEvaluationDepth is decremented. |

| **CurrentEvaluationDepth** |
| PURPOSE: The current “depth” of evaluation. This value is used for the purposes of garbage collection. At the beginning of the execution of each rule, deffunction, generic function, or message-handler, this value is incremented by one. At the |
completion of the execution of each rule, deffunction, generic function, or message-handler, this value is decremented by one. Note that the execution of a system or user-defined function does not affect this value.

<table>
<thead>
<tr>
<th><strong>EvaluationError</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Boolean flag which indicates if an error has occurred while evaluating an expression.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>HaltExecution</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Boolean flag which indicates if execution (rules, certain functions such as while, deffunctions, etc.) should be halted.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CurrentExpression</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: As expressions are evaluated, maintains list of arguments for each expression evaluation.</td>
</tr>
</tbody>
</table>

GLOBAL FUNCTIONS

**Argument Access Functions**

Purpose: A series of functions which allows access to the arguments of an expression. Some access functions are implemented as macros. The following are access functions implemented as functions: `RtnArgCount`, `ArgCountCheck`, `ArgTypeCheck`, `RtnLong`, `RtnUnknown`, `RtnLexeme`, `RtnDouble`, and `ArgRangeCheck`. See the Advanced Programming Guide for further details.

**CLIPSFunctionCall**

Purpose: Allows functions external to CLIPS to execute function calls. See the Advanced Programming Guide for further details.

**DefineFunction**

Purpose: Defines a function to be accessible to CLIPS.

Arguments: Function access name, pointer to the function, type of return value, and actual function name.

**EvaluateExpression**

Purpose: Evaluates an expression.
ARGUMENTS: An expression to evaluate, and a pointer to a data structure in which to return a value.

RETURNS: The current value of **EvaluationError**. The return value of the expression is stored in the data structure.

---

### GetBoundVariable

**PURPOSE:** Searches the **BindList** for a specified variable.

**ARGUMENTS:** The name of the variable and a pointer to a DATA_OBJECT structure in which to store variable, if found.

**RETURNS:** A boolean value. TRUE if the variable was found, otherwise FALSE.

---

### GetEvaluationError

**PURPOSE:** Returns the **EvaluationError** flag.

---

### GetHaltExecution

**PURPOSE:** Returns the **HaltExecution** flag.

---

### PrintDataObject

**PURPOSE:** Prints a DATA_OBJECT structure to the specified logical name.

**ARGUMENTS:** A pointer to a DATA_OBJECT structure and a logical name.

---

### PropagateReturnValue

**PURPOSE:** Decrements the associated depth for a value stored in a DATA_OBJECT structure. In effect, the values returned by certain evaluations (such as a deffunction call) are passed up to the previous depth of evaluation. The return value's depth is decremented so that it will not be garbage collected along with other items that are no longer needed from the evaluation that generated the return value.

**ARGUMENTS:** A pointer to a DATA_OBJECT structure.

---

### ReturnValues

**PURPOSE:** Returns a linked list of DATA_OBJECT structures to the pool of free memory.

**ARGUMENTS:** A pointer to the head DATA_OBJECT structure in a list.
Return Value Access Functions

PURPOSE: A series of functions which allows access to the return value data structures. Most of these access functions are implemented as macros. See the Advanced Programming Guide for further details.

SetEvaluationError

PURPOSE: Sets the EvaluationError flag.

ARGUMENTS: A boolean value (the new value of the flag). If the value of the flag is TRUE, then the HaltExecution flag is also set to TRUE.

SetHaltExecution

PURPOSE: Sets the HaltExecution flag.

ARGUMENTS: A boolean value (the new value of the flag).

SetMultifieldErrorValue

PURPOSE: Creates a multifield value of length zero for error returns.

ARGUMENTS: A pointer to a DATA_OBJECT structure in which the error value is to be stored.

ValueDeinstall

PURPOSE: Decrement the appropriate count (in use) values for a DATA_OBJECT structure.

ARGUMENTS: A pointer to a DATA_OBJECT structure.

ValueInstall

PURPOSE: Increments the appropriate count (in use) values for a DATA_OBJECT structure.

ARGUMENTS: A pointer to a DATA_OBJECT structure.

INTERNAL FUNCTIONS

NonexistantError

PURPOSE: Prints the error message for a nonexistant argument.
ARGUMENTS: The name of the access function which couldn’t find the argument, the name of the function which called the access function, and the index position of the argument requested.

**WrongTypeError**

PURPOSE: Prints the error message for the wrong type of argument.

ARGUMENTS: The name of the access function which couldn’t find the argument, the name of the function which called the access function, and the name of the type expected.
Command Line Module

The Command Line Module (commline.c) contains the basic functions for setting up a simple command line processor for CLIPS commands.

Command line routines are oriented for building interfaces that use an event-driven philosophy. These interfaces have windows, menus, and/or command entry windows. In an event-driven interface, keyboard input is just one of several possible events. If a key is pressed, it is placed in an input buffer. The input buffer will not be processed until a complete command has been entered. In CLIPS, commands are delimited by a set of matching parentheses. In addition, commands may also be variables or constants. During entry to the input buffer, other events such as menu selections can also be processed because CLIPS has not yet begun to process the input command.

The basic input buffer to CLIPS should be used for accepting keyboard entry. File entry should be permitted to lock out other events. In effect, menu commands cannot be accessed during the loading of a file. The file loading operation represents a complete event by itself whereas a single keyboard character entry is a single event.

The basic command loop for CLIPS works as follows:

```plaintext
procedure CommandLoop
   print the CLIPS prompt
   do forever
      call EventFunction
      if a complete command is in the input buffer then
         perform the command
         clear the input buffer
         print the CLIPS prompt
      end if
   end do
end procedure
```

Notice that the loop calls the `EventFunction` procedure repeatedly. The command is executed only when the `CompleteCommand` function indicates that a complete command is waiting in the input buffer. A typical `EventFunction` procedure for a non-windowed interface would be

```plaintext
procedure GenericEventFunction
   get a character from the keyboard
   stuff the character into the input buffer
end procedure
```

The only event possible is to grab a character which is then stuffed into an input buffer. If a command has been completed, the `CompleteCommand` function returns a non-zero value. Simple modifications to this basic function allow for the easy operation of a windowed interface as shown following.
procedure WindowEventFunction
get the next event
if the event is a key press then
stuff the character into the input buffer
else if the event is a menu selection
execute the menu selection
else if the event is a window operation
execute the window operation
end if
end procedure

In this function, a routine is used to get the next event. Depending upon the exact nature of the event, different actions are taken. This type of setup will allow the user to begin entering a command, browse the data base using menu options, and then finish entering the command.

GLOBAL VARIABLES

<table>
<thead>
<tr>
<th>EvaluatingTopLevelCommand</th>
</tr>
</thead>
</table>
PURPOSE: Boolean flag which indicates whether a top-level command is currently being executed.

INTERNAL VARIABLES

<table>
<thead>
<tr>
<th>CommandString</th>
</tr>
</thead>
</table>
PURPOSE: Input buffer for the command string being formed.

<table>
<thead>
<tr>
<th>EventFunction</th>
</tr>
</thead>
</table>
PURPOSE: A pointer to the function to be called to process the next event.

<table>
<thead>
<tr>
<th>MaximumCharacters</th>
</tr>
</thead>
</table>
PURPOSE: Current maximum length of the CommandString.

<table>
<thead>
<tr>
<th>MemoryStatusFunction</th>
</tr>
</thead>
</table>
PURPOSE: A pointer to a function which is periodically called during the command loop to allow the interface to update a display which indicates the amount of memory used by CLIPS.

<table>
<thead>
<tr>
<th>ParsingTopLevelCommand</th>
</tr>
</thead>
</table>
PURPOSE: Boolean flag which indicates whether a top-level command is currently being parsed.
**VersionString**

**PURPOSE:** The character string that is printed when CLIPS first starts indicating the CLIPS version number and date of creation.

**GLOBAL FUNCTIONS**

**AppendCommandString**

**PURPOSE:** Appends a value to the contents of the CommandString.

**ARGUMENTS:** A string.

**CommandLoop**

**PURPOSE:** Endless loop which waits for user commands and then executes them. The command loop will bypass the EventFunction if there is an active batch file.

**CompleteCommand**

**PURPOSE:** Determines whether a string forms a complete command. A complete command is either a constant, a variable, or a function call which is followed (at some pointer) by a carriage return. Once a complete command is found (not including the parenthesis), extraneous parenthesis and other tokens are ignored.

**ARGUMENTS:** A string.

**RETURNS:** Integer value. 1 if the string forms a complete command without errors, 0 if the string forms an incomplete command without errors, and -1 if the string has errors (e.g., the command begins with a right parenthesis).

**OTHER NOTES:** Implemented as several functions.

**ExpandCommandString**

**PURPOSE:** Appends a character to the CommandString.

**ARGUMENTS:** Character to be appended. This routine properly handles the backspace character by removing a character from the CommandString.

**FlushCommandString**

**PURPOSE:** Empties the contents of the CommandString.
**GetCommandString**

PURPOSE: Returns a pointer to the contents of the `CommandString`.

RETURNS: Current `CommandString`.

**PrintPrompt**

PURPOSE: Prints the CLIPS command prompt.

**RouteCommand**

PURPOSE: Processes a completed command.

ARGUMENTS: A command string.

RETURNS: Boolean value. TRUE if the command was successfully executed, otherwise FALSE.

OTHER NOTES: Creates a string router with its command string argument and then calls the appropriate parsing and execution functions to process the command.

**SetCommandString**

PURPOSE: Sets the contents of the `CommandString` to a specific value.

ARGUMENTS: A string.

OTHER NOTES: Flushes current contents of the `CommandString`.

**SetEventFunction**

PURPOSE: Replaces the current value of `EventFunction`.

ARGUMENTS: A pointer to the new event-handling function.

RETURNS: A pointer to the old event-handling function.

**SetMemoryStatusFunction**

PURPOSE: Replaces the current value of `MemoryStatusFunction`.

ARGUMENTS: A pointer to the new memory status function.

**TopLevelCommand**

PURPOSE: Indicates whether a top-level command is being parsed.
RETURNS: Returns the value of ParsingTopLevelCommand.

INTERNAL FUNCTIONS

**DefaultGetNextEvent**

PURPOSE: Default event-handling function. Handles only keyboard events by first calling GetcCLIPS to get a character and then calling ExpandCommandString to add the character to the CommandString.

**DoComment**

PURPOSE: Skips over a comment contained within a string until a line feed or carriage return is encountered.

ARGUMENTS: A pointer to a string and an integer representing the position of the character in the string currently being scanned.

RETURNS: An integer. The character position in the string where the comment terminates.

**DoString**

PURPOSE: Skips over a string contained within a string until the closing quotation mark is encountered.

ARGUMENTS: A pointer to a string, an integer representing the position of the character in the string currently being scanned, and a pointer to an integer flag which indicates if the closing quotation mark was actually encountered.

RETURNS: An integer. The character position in the string where the string terminates. If the string is terminated by a quotation mark then the integer flag passed as an argument is set to TRUE, otherwise the flag is set to FALSE.

**DoWhiteSpace**

PURPOSE: Skips over white space consisting of spaces, tabs, and form feeds that is contained within a string.

ARGUMENTS: A pointer to a string and an integer representing the position of the character in the string currently being scanned.

RETURNS: An integer. The character position in the string where the white space terminates.
Several defining constructs appear in CLIPS: defrule, deffacts, deftemplate, defglobal, deffunction, defclass, definstances, defmessage-handler, defgeneric, and defmethod. The Construct Manager Module (constrct.c) handles a variety of operations generic to these constructs.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

BeforeClearFunction

PURPOSE: Contains a pointer to a function which is to be called before the clear command is executed (if this variable is not NULL). If the function returns FALSE, the clear command is not performed.

BeforeResetFunction

PURPOSE: Contains a pointer to a function which is to be called before the reset command is executed (if this variable is not NULL). If the function returns FALSE, the reset command is not performed.

Executing

PURPOSE: Boolean flag. If TRUE, indicates that a construct is being executed.

ListOfClearFunctions

PURPOSE: Contains a list of functions to be called whenever a clear command is issued.

ListOfConstructs

PURPOSE: Contains the list of constructs recognized by CLIPS along with pointers to the functions which parse each construct.

ListOfResetFunctions

PURPOSE: Contains a list of functions to be called when a reset is performed.
**ListOfSaveFunctions**

**PURPOSE:** Contains list of functions to be called whenever a save command is issued.

**PrintWhileLoading**

**PURPOSE:** Boolean flag. If on, then loading information will be printed during the loading of constructs. If off, then no loading information is printed. The top-level `load` command enables this flag (and either single characters or a more lengthy message will be printed for each construct depending upon the value of `WatchCompilations`). The embedded `LoadConstructs` function does not set this flag and thus by default messages will not be printed when this routine is called.

**WatchCompilations**

**PURPOSE:** Boolean flag. If on, indicates that the progress of construct definitions should be displayed.

**GLOBAL FUNCTIONS**

**AddClearFunction**

**PURPOSE:** Adds a function to the `ListOfClearFunctions`.

**ARGUMENTS:** A name to be associated with the function, a pointer to the function, and the priority of the clear item.

**AddConstruct**

**PURPOSE:** Adds a construct and its associated parsing function to the `ListOfConstructs`.

**ARGUMENTS:** Name of construct for which the parsing function is to be applied, and a pointer to the parsing function.

**AddResetFunction**

**PURPOSE:** Adds a function to `ListOfResetFunctions`.

**ARGUMENTS:** A name to be associated with the function, a pointer to the function, and the priority of the reset item.
AddSaveFunction

PURPOSE: Adds a function to theListOfSaveFunctions.

ARGUMENTS: A name to be associated with the function and a pointer to the function.

CallClearFunctions

PURPOSE: Calls all clear functions in theListOfClearFunctions.

ClearCLIPS

PURPOSE: Clears the CLIPS environment. See the Basic Programming Guide for details on the effects of a clear.

OTHER NOTES: Calls the BeforeClearFunction, then each function in theListOfClearFunctions in order of descending priority.

ExecutingConstruct

PURPOSE: Returns the value of Executing.

GetCompilationsWatch

PURPOSE: Returns the value of WatchCompilations.

GetPrintWhileLoading

PURPOSE: Returns the value of PrintWhileLoading.

InitializeConstructs

PURPOSE: Initializes the Construct Manager.

InitializeIgnoredConstructs

PURPOSE: Initializes some parsing routines for skipping over CRSV constructs not handled by CLIPS such as defrelation and defexternal.

LoadConstructs

PURPOSE: Loads a set of constructs into the current CLIPS environment from a file.

ARGUMENTS: A file name.
OTHER NOTES: Converts file name to a logical name and calls function `LoadConstructsFromLogicalName`.

**LoadConstructsFromLogicalName**

PURPOSE: Loads a set of constructs into the current CLIPS environment from a specified logical name.

ARGUMENTS: A logical name.

OTHER NOTES: Calls function `ParseConstruct` to read in each construct.

**ParseIgnoredConstruct**

PURPOSE: Parsing routine for skipping over constructs recognized, but not handled by CLIPS (such as CRSV constructs).

ARGUMENTS: Logical name from which input is read.

**ParseConstruct**

PURPOSE: Parses a construct.

ARGUMENTS: Name of construct to be parsed, and a logical name from which input is to be read.

RETURNS: An integer. -1 if the construct name has no parsing function, 0 if the construct was parsed successfully, and 1 if the construct was parsed unsuccessfully.

OTHER NOTES: Construct parsing functions should return a value of 0 if the construct is parsed successfully and a value of 1 if the construct is not parsed successfully.

**RemoveClearFunction**

PURPOSE: Removes a function from the `ListOfClearFunctions`.

ARGUMENTS: Name associated with the function.

**RemoveConstruct**

PURPOSE: Removes a construct and its associated parsing function from the `ListOfConstructs`.

ARGUMENTS: Name of construct to be removed.
RemoveResetFunction

PURPOSE: Removes a function from the ListOfResetFunctions.

ARGUMENTS: Name associated with reset function.

RemoveSaveFunction

PURPOSE: Removes a function from the ListOfSaveFunctions.

ARGUMENTS: Name associated with function.

ResetCLIPS

PURPOSE: Resets the CLIPS environment. See the Basic Programming Guide for details on the effects of a reset.

OTHER NOTES: Calls the BeforeResetFunction, then each function in the ListOfResetFunctions in order of descending priority.

SaveConstructs

PURPOSE: Saves the constructs currently in the CLIPS environment to a file. This function is the primary routine called by the save command.

ARGUMENTS: The name of the file to which constructs should be saved.

OTHER NOTES: Opens the specified file then calls each function in the ListOfSaveFunctions.

SetBeforeClearFunction

PURPOSE: Sets the value of BeforeClearFunction.

ARGUMENTS: A pointer to a function.

SetBeforeResetFunction

PURPOSE: Sets the value of BeforeResetFunction.

ARGUMENTS: A pointer to a function.

SetCompilationsWatch

PURPOSE: Sets the value of WatchCompilations.

ARGUMENTS: A boolean value (TRUE or FALSE).
SetExecutingConstruct

PURPOSE: Sets the value of Executing.
ARGUMENTS: A boolean value (TRUE or FALSE).

SetPrintWhileLoading

PURPOSE: Sets the value of PrintWhileLoading.
ARGUMENTS: A boolean value (TRUE or FALSE).

ValidConstruct

PURPOSE: Determines whether a construct is in theListOfConstructs.
ARGUMENTS: Name of the construct.
RETURNS: Boolean value. TRUE if the construct has a parsing function; otherwise FALSE.

INTERNAL FUNCTIONS

ErrorAlignment

PURPOSE: Positions the parser at a token which indicates the beginning of a valid construct. If called as the result of an error in a construct, this routine skips over tokens until it finds the beginning of a new construct. If an error hasn't occurred, then this routine checks to see that the parser is currently at the beginning of a new construct (a left parenthesis followed by a constructs name).

ARGUMENTS: Logical name from which input was being read when an error was detected, a boolean value indicating whether an error has occurred, and a pointer to a data structure in which parsed tokens can be stored.
Utility Module

The Utility Module (utility.c) contains a number of generally useful functions including functions for printing primitive data types, constructing string representations of primitive data types, appending characters and strings to other strings, checking argument types, printing generic error and informational messages, adding and manipulating items that can be watched using the `watch` command, and performing periodic garbage collection.

The method CLIPS uses to perform periodic garbage collection merits some discussion. Garbage collection and ephemeral “items” have already been discussed to some extent as they relate to the Symbol Module (symbol.c) and the Evaluation Module (evaluatn.c). Garbage within CLIPS comes in several varieties. The first variety is garbage that can be immediately discarded when it is no longer in use. As an example of this, consider the following command sequence.

```
CLIPS> (open "temp.txt" temp "w")
TRUE
CLIPS> (printout temp "Hello World" crlf)
CLIPS> (close temp)
TRUE
CLIPS>
```

When the file "temp.txt" is opened using the `open` command, data structures are allocated which associate the logical name temp with the newly opened file. When the `close` command is used to close the file, the data structure previously allocated are no longer needed and can be immediately returned to the pool of free memory. In this case, garbage collection occurs for the data structures at the same time the garbage is created. Deleting constructs is another example of this type of garbage collection since the memory used by these constructs is almost always immediately returned to the pool of free memory (with some exceptions such as deleting an executing rule).

The second type of garbage collection occurs when an item appears to be garbage (but it cannot yet be determined), or an item is garbage but is temporarily being referred to by another data structure. As an example of this, consider the following command sequence.

```
CLIPS> (assert (colors red green))
CLIPS> (defrule remove-fact
    ?f <- (colors ?x ?y)
    =>
    (retract ?f)
    (printout t "Colors: " ?x " " ?y crlf))
CLIPS> (run)
CLIPS>
```

When the fact (color red green) is retracted by the `remove-fact` rule, the symbols `red` and `green` become garbage since they are no longer permanently referred to by any data structure. However, these values are still needed for the `printout` command which
follows the retract command so the values cannot be garbage collected just yet. Once the rule has completed execution the values can be safely garbage collected.

Garbage collection can occur at the completion of each rule, deffunction, generic function, or message-handler that is executed, however, it does not always occur each time one of these boundaries is encountered. CLIPS uses some heuristics to determine if garbage collection should actually take place. First, either the size of number of items subject to garbage collection must exceed a specified value. That is, CLIPS will not garbage collect to reclaim 120 bytes of memory. Second, if garbage collection does not free enough memory at a specified evaluation depth, then garbage collection at that depth will not be repeated until a larger amount of garbage has been created. This prevents garbage collection from being repeatedly attempted on items that cannot yet be freed.

GLOBAL VARIABLES

AddressesToStrings

PURPOSE: Boolean flag which indicates whether addresses (external, fact, or instance) should be printed using the notation for addresses or should be printed with quotes surrounding them. This is used by functions such as save-facts which are not capable of reloading addresses and so must convert the addresses to a safe form.

CurrentEphemeralCountMax

PURPOSE: The current maximum number of ephemeral items allowed before periodic garbage collection is attempted.

CurrentEphemeralSizeMax

PURPOSE: The current maximum amount of memory used by ephemeral items before periodic garbage collection is attempted.

EphemeralItemCount

PURPOSE: The current number of “items” that can be potentially garbage collected.

EphemeralItemSize

PURPOSE: The amount of memory used by all of the “items” that can be potentially garbage collected.
**PreserveEscapedCharacters**

PURPOSE: Boolean flag which indicates whether the backslash escape character should be reembedded within a string when the string is printed.

**INTERNAL VARIABLES**

**ListOfCleanupFunctions**

PURPOSE: Contains a list of functions to be called when a periodic cleanup is performed.

**ListOfPeriodicFunctions**

PURPOSE: Contains a list of functions to be called when a periodic cleanup is checked. These function are always called whenever `PeriodicCleanup` is called. The `ListOfCleanupFunction` is only called if the cleanup heuristics indicate that a periodic cleanup should be performed. These functions are useful for updating displays or checking for events in machine specific interfaces layered on top of the CLIPS kernel.

**ListOfWatchItems**

PURPOSE: Contains a list of structures that represent the items that can be watched using the `watch` command.

**GLOBAL FUNCTIONS**

**AddCleanupFunction**

PURPOSE: Adds a function to the `ListOfCleanupFunctions`.

ARGUMENTS: A name to be associated with the function, a pointer to the function, and the priority of the cleanup item.

**AddPeriodicFunction**

PURPOSE: Adds a function to the `ListOfPeriodicFunctions`.

ARGUMENTS: A name to be associated with the function, a pointer to the function, and the priority of the periodic item.
AddWatchItem

PURPOSE: Adds a watch item to the ListOfWatchItems.

ARGUMENTS: The name of the watch item, a pointer to the integer in which the watch item's value is stored, and the priority of the watch item.

AppendNToString

PURPOSE: Appends a specified number of characters from one string to another. Expands the appended string, if necessary, to create enough space.

ARGUMENTS: A pointer to the appending string, a pointer to the string to be appended, a pointer to the current length of the appended string, a pointer to the maximum length that the appended string can contain, and the maximum number of characters that are to be appended to the string.

RETURNS: The new appended string. The string that is appended may be dynamically reallocated to create a larger string. The current length and maximum length values are updated by this routine.

AppendToString

PURPOSE: Appends one string to another. Expands the appended string, if necessary, to create enough space.

ARGUMENTS: A pointer to the appending string, a pointer to the string to be appended, a pointer to the current length of the appended string, and a pointer to the maximum length that the appended string can contain.

RETURNS: The new appended string. The string that is appended may be dynamically reallocated to create a larger string. The current length and maximum length values are updated by this routine.

AtomDeinstall

PURPOSE: Decrements the count value for a single primitive data type.

ARGUMENTS: The type of the primitive data type and the value of the primitive data type.
**AtomInstall**

**PURPOSE:** Increments the count value for a single primitive data type.

**ARGUMENTS:** The type of the primitive data type and the value of the primitive data type.

**GetConstructName**

**PURPOSE:** Checks for an appropriate symbolic name as the argument to a function call during run-time. Used by functions such as `ppdefrule` and `undefrule` which require a symbolic value as the name of a defrule. A name must be a symbol, not a string.

**ARGUMENTS:** Expected number of arguments, name of function being executed, position in the argument where the name should occur, and a string describing the name type being sought (i.e., "defrule name", "deffacts name").

**RETURNS:** Returns the symbolic value (a string) found in the position. If an error occurs, returns NULL.

**CLIPSSystemError**

**PURPOSE:** Standard error message used to indicate that a CLIPS internal error has been detected.

**ARGUMENTS:** A string indicating the module in which the error was detected and an ID number associated with error.

**ExpandStringWithChar**

**PURPOSE:** Adds a character to a string, expanding the string if necessary.

**ARGUMENTS:** Character to be added, destination string, a pointer to the integer representing the insertion point in the string, a pointer to the integer representing the maximum size of the string, and new size for the string if it must be expanded.

**RETURNS:** A string with the character added to it. The string that is returned may have been dynamically reallocated to create a larger string. The current length and maximum length values are updated by this routine.
## ExpectedTypeError

**PURPOSE:** Standard error message used when wrong type of argument has been used in an expression.

**ARGUMENTS:** Name of function, position of argument, and string containing a description of the expected type.

## ExpectedCountError

**PURPOSE:** Standard error message used when the wrong number of arguments has been used in the argument list of a function call.

**ARGUMENTS:** Name of function, relation value for arguments being checked (EXACTLY, AT LEAST, NO MORE THAN), and comparison value for arguments being checked.

## FloatToString

**PURPOSE:** Converts a float to a string using the CLIPS numeric format. CLIPS uses the %g format option from the C library routine `sprintf` to print floating point numbers. This format selects either scientific notation or prints all the digits of the number (whichever ends up taking less space). In addition, CLIPS makes sure that all floats are printed with at least one digit following the decimal point.

**ARGUMENTS:** A floating-point number.

**RETURNS:** A string.

**OTHER NOTES:** Return value is stored in a static data area. Subsequent calls to this function will write over this data area. If the return value must be stored, it should be duplicated.

## GetFileName

**PURPOSE:** Checks for an appropriate file name as the argument of a function call during run-time. A file name must be a string or a symbol.

**ARGUMENTS:** Name of function being executed and the position of the argument in the argument list that contains the file name.

**RETURNS:** File name.
GetLogicalName

PURPOSE: Checks for an appropriate logical name in an expression during run-time.

ARGUMENTS: The position of the argument in the argument list that contains the logical name and the logical name to be used if the default logical name, t, is found.

RETURNS: A string representing the logical name. If found, the value designated as the default logical name is returned. Returns NULL if the argument is unacceptable as a logical name.

GetNthWatchName

PURPOSE: Given an index, returns the name of the nth item in the ListOfWatchItems (which is useful for constructing a menu).

ARGUMENTS: An integer index.

RETURNS: The name of the nth watch item (a character string).

GetNthWatchValue

PURPOSE: Given an index, returns the value of the nth item in the ListOfWatchItems.

ARGUMENTS: An integer index.

RETURNS: The boolean value of the nth watch item.

GetWatchItem

PURPOSE: Returns the value of a watch item.

ARGUMENTS: The name of the watch item.

LongIntegerToString

PURPOSE: Converts a long integer to a string.

ARGUMENTS: A long integer.

RETURNS: A string.

OTHER NOTES: Return value is stored in a static data area. Subsequent calls to this function will write over this data area. If return value must be stored, it should be duplicated.
**OpenErrorMessage**

**PURPOSE:** Standard error message used when a function cannot open a file.

**ARGUMENTS:** The name of the function and the file name that could not be opened.

**PeriodicCleanup**

**PURPOSE:** Returns ephemeral garbage to the pool of free memory. When this function is called and it is determined that there is sufficient garbage to warrant a cleanup, then each of the functions in the **ListOfCleanupFunctions** will be called to perform garbage collection.

**ARGUMENTS:** Two boolean values. The first value indicates whether all evaluation depths should cleaned up. Normally, garbage collection only occurs for items that have an evaluation depth greater than the current evaluation depth. If this boolean argument is TRUE, however, the current evaluation depth will be temporarily set to a value which forces garbage collection for all depths. The second boolean value is used to determine whether heuristics are used in performing the garbage collection.

**PrintAtom**

**PURPOSE:** Prints a CLIPS primitive data type (which does not include multifield values).

**ARGUMENTS:** Logical name to which output is sent, the type of the primitive data type, and the value of the primitive data type.

**PrintFloat**

**PURPOSE:** Prints a number to a logical name using the CLIPS print format for numbers.

**ARGUMENTS:** A floating-point number and a logical name.

**PrintInChunks**

**PURPOSE:** Prints a string in chunks to accommodate systems which have a limit on the maximum size of a string which can be printed.

**ARGUMENTS:** String to be printed and logical name to which the string is to be printed.
### PrintLongInteger

**PURPOSE:** Prints a long integer to a logical name using CLIPS print format for numbers.

**ARGUMENTS:** A long integer and a logical name.

### PrintTally

**PURPOSE:** Standard message for functions which print a message indicating the number of items displayed (e.g. the `facts` command).

**ARGUMENTS:** The logical name to which output is to be sent, the number of items tallied, and singular and plural strings for the items tallied (e.g. "fact" and "facts").

**OTHER NOTES:** No message is printed if the number of items tallied is zero.

### RemoveCleanupFunction

**PURPOSE:** Removes a function from the `ListOfCleanupFunctions`.

**ARGUMENTS:** Name associated with the cleanup item.

### RemovePeriodicFunction

**PURPOSE:** Removes a function from the `ListOfPeriodicFunctions`.

**ARGUMENTS:** Name associated with the periodic item.

### RestoreAllWatchItems

**PURPOSE:** Restores the old value of each watch items that was saved when the `SaveAllWatchItems` function was called.

### SetAllWatchItems

**PURPOSE:** Sets all of the watch items to a particular value and remembers the old value of each watch item.

**ARGUMENTS:** The new boolean value to which all watch items are set.

### SetWatchItem

**PURPOSE:** Sets the value of a watch item.
ARGUMENTS: The name of the watch item and the new boolean value. The string “all” may be used to set all watch items to a particular value.

SyntaxErrorMessage

PURPOSE: Standard error message used for syntax errors.
ARGUMENTS: The type of syntax error that occurred (e.g. “defrule”, “conditional elements”, etc.).

INTERNAL FUNCTIONS

AddCPFunction

PURPOSE: Driver routine for implementing the functions AddCleanupFunction and AddPeriodicFunction.
ARGUMENTS: A name to be associated with the function, a pointer to the function, the priority of the item, and a pointer to a pointer to the list to which the function is to be added.

RemoveCPFunction

PURPOSE: Driver routine for implementing the functions RemoveCleanupFunction and RemovePeriodicFunction.
ARGUMENTS: Name associated with the periodic item and a pointer to a pointer to the list from which the function is to be removed.
Fact Manager Module

The Fact Manager Module (factmngr.c) provides the necessary functionality to maintain, update, and browse facts. It also provides top-level implementation of the assert and retract commands. Functions for displaying and browsing facts are likewise provided. The other major functionality provided by this module is a hash table containing facts. Unlike OPS5, the CLIPS inferencing paradigm does not allow two occurrences of the same fact to be in the fact-list. A fact hash table provides a convenient method for determining if a fact is already in the fact-list.

GLOBAL VARIABLES

**ChangeToFactList**

PURPOSE: Boolean flag. If TRUE, indicates that the FactList has been altered. Updates to TRUE whenever a fact is asserted or retracted.

INTERNAL VARIABLES

**AssertRetractInProgress**

PURPOSE: Boolean flag. If TRUE, an assertion or retraction of a fact is currently occurring.

**FactDuplication**

PURPOSE: Boolean flag. If TRUE, duplications of facts are allowed in the FactList.

**FactHashTable**

PURPOSE: Stores all facts used by CLIPS.

C IMPLEMENTATION: Implemented as an array. Each entry corresponds to a list of fact table entries. Collisions are resolved by adding the fact entry to the list of entries.

OTHER NOTES: Information about facts is also stored in the FactList. Used primarily to quickly determine if a fact is already in the FactList.

**FactList**

PURPOSE: Stores all facts used by CLIPS.

C IMPLEMENTATION: Implemented as a list.
### GarbageFacts

**PURPOSE:** Points to the list of facts that can be returned to free memory. These facts have typically been retracted but need to remain in memory because there are outstanding references to them (e.g., they are needed for the duration of a rule firing in case a variable access within the fact is made or other facts might refer to them).

### LastFact

**PURPOSE:** Points to the last fact in the FactList.

### ListOfSegments

**PURPOSE:** Contains the list of multifield values that have been dynamically created.

### NextFactIndex

**PURPOSE:** Long Integer value to be used as the fact-index for the next asserted fact.

### NumberOfFacts

**PURPOSE:** Contains an integer count of the number of facts in the FactList.

### WatchFacts

**PURPOSE:** Boolean flag. If TRUE, indicates that fact assertions and rejections should be displayed.

### GLOBAL FUNCTIONS

#### AddFact

**PURPOSE:** Coordinates assertion of a fact into the FactList.

**ARGUMENTS:** A fact.

#### AddHashedFact

**PURPOSE:** Adds a fact to the FactHashTable.

**ARGUMENTS:** A fact and the hash value of that fact.
OTHER NOTES: Does not check to determine if the fact is already in the FactHashTable.

**AddToSegmentList**

PURPOSE: Adds a fact to the ListOfSegments. Can be used in conjunction with the CreateFact function to perform the same functionality as CreateMultifield.

ARGUMENTS: A pointer to a fact.

**AssertString**

PURPOSE: Converts a string to a fact and then asserts it. Uses the functions StringToFact and AddFact.

ARGUMENTS: A string.

RETURNS: A pointer to the newly asserted fact.

**CreateFact**

PURPOSE: Allocates the data structures necessary for a fact containing a specified number of fields.

ARGUMENTS: Number of fields in the fact.

RETURNS: A fact of the appropriate size.

**CreateMultifield**

PURPOSE: Allocates the data structures necessary for a multifield containing the specified number of fields and adds the newly created multifield to the ListOfSegments.

ARGUMENTS: Number of fields in the multifield.

RETURNS: A multifield of the appropriate size. Note that the structures used for the multifields are identical to the fact structures.

**DecrementFactCount**

PURPOSE: Decrements the count value for a fact.

ARGUMENTS: A pointer to a fact.
**DuplicateSegment**

PURPOSE: Copies the contents of a multifield value to another multifield value.

ARGUMENTS: A pointer to the source multifield and a pointer to the destination multifield.

**FactCompare**

PURPOSE: Determines if two facts are identical.

ARGUMENTS: Two facts.

RETURNS: Boolean value. True if facts are identical; otherwise false.

**FactDeinstall**

PURPOSE: Called when a fact is garbage collected (not when it is retracted). Decrements the `NumberOfFacts` and calls `SegmentDeinstall`.

ARGUMENTS: A fact.

**FactExists**

PURPOSE: Determines if a fact exists in the `FactHashTable`.

ARGUMENTS: A fact and the hash value of that fact.

RETURNS: A pointer to the fact in the `FactHashTable` if it already exists, otherwise NULL.

**FactInstall**

PURPOSE: Called when a fact is newly created. Increments the `NumberOfFacts` and calls `SegmentInstall`.

ARGUMENTS: A fact.

**FindIndexedFact**

PURPOSE: Finds a fact by fact-index.

ARGUMENTS: The fact-index of the fact being sought.

RETURNS: A pointer to the fact with the specified fact-index or NULL if a fact with the specified fact-index does not exist.
FlushSegments

PURPOSE: Removes any multifield values from the ListOfSegments that have a zero count and an evaluation depth greater than the current evaluation depth.

ARGUMENTS: A fact.

GetFactDuplication

PURPOSE: Returns the current value of the FactDuplication flag.

RETURNS: A boolean value.

GetFactIndex

PURPOSE: Returns fact-index associated with a fact.

ARGUMENTS: A pointer to a fact.

RETURNS: The fact-index of the fact (an integer value).

GetFactListChanged

PURPOSE: Returns the value of ChangeToFactList.

GetFactPPForm

PURPOSE: Returns the pretty print representation of a fact.

ARGUMENTS: A pointer to a fact, a pointer to a buffer in which to store the pretty print representation, and the size of the buffer.

RETURNS: No return value. The buffer passed as an argument is used to store the pretty print representation.

GetNextFact

PURPOSE: Returns a pointer to the “next” fact in the FactList.

ARGUMENTS: A pointer to a fact in the FactList.

RETURNS: Next fact after the fact passed as an argument. If a NULL pointer is used, the first fact in the FactList is returned.

GetNumberOfFacts

PURPOSE: Returns the value of the NumberOfFacts.
RETURNS: An integer value.

### HashFact

**PURPOSE:** Computes a hash value for a fact.

**ARGUMENTS:** A fact.

**RETURNS:** An integer hash value less than the array size of the FactHashTable.

### IncrementFactCount

**PURPOSE:** Increments the count value for a fact.

**ARGUMENTS:** A pointer to a fact.

### InitializeFacts

**PURPOSE:** Performs all necessary initialization for facts (initializing the FactHashTable, adding reset and clear functions, adding the facts watch item, and calling DefineFunction to add fact related commands).

### ListFacts

**PURPOSE:** Displays all of the fact in the FactList to the logical name wdisplay.

### PrintFact

**PURPOSE:** Displays the fields of a fact enclosed within parentheses.

**ARGUMENTS:** A fact and logical name to which output is to be sent.

### PrintFactWithIdentifier

**PURPOSE:** Displays the fact-index of a fact followed by the fact. Uses the function PrintFact.

**ARGUMENTS:** A fact and logical name to which output is to be sent.

### RemoveAllFacts

**PURPOSE:** Removes all facts from the FactList.

### RemoveHashedFact

**PURPOSE:** Removes a fact from the FactHashTable.
ARGUMENTS: A fact.

**RemoveOldFacts**

PURPOSE: Returns facts in the list of GarbageFacts to the memory manager. Facts are only returned if there are no outstanding references to them (e.g. they are not being used by the currently executing rule or other facts do not refer to them) and the evaluation depth at which they were created is greater than the current evaluation depth.

**RetractFact**

PURPOSE: Coordinates retraction of a fact from the FactList.

ARGUMENTS: A fact.

**ReturnElements**

PURPOSE: Returns the data structures associated either with a fact or a multifield to the memory manager.

ARGUMENTS: A fact or a multifield (both use the same structures).

OTHER NOTES: Fact or multifield should be deinstalled using FactDeinstall or SegmentDeinstall respectively before removal.

**SegmentDeinstall**

PURPOSE: Decrements count values for the constant values (symbols, strings, integers, floats, etc.) found in a multifield value. Decrements the number of references to the multifield by one.

ARGUMENTS: A multifield (which is stored using fact data structures).

**SegmentInstall**

PURPOSE: Increments count values for the constant values (symbols, strings, integers, floats, etc.) found in a multifield value. Increments the number of references to the multifield by one.

ARGUMENTS: A multifield (which is stored using fact data structures).

**SetFactDuplication**

PURPOSE: Sets the current value of the FactDuplication flag.

ARGUMENTS: A boolean value (the new value of the flag).
RETURNS: A boolean value (the old value of the flag).

### SetFactID

PURPOSE: Sets the value of NextFactID.
ARGUMENTS: An integer.

### SetFactListChanged

PURPOSE: Sets value of ChangeToFactList.
ARGUMENTS: Boolean value.

### StringToFact

PURPOSE: Parses a string and converts it to a fact. The string should be a series of constants and should not contain enclosing parentheses.
ARGUMENTS: A string.
RETURNS: A pointer to the newly created fact.

### StringToMultifield

PURPOSE: Parses a string and converts it to a multifield value. The string should be a series of constants and should not contain enclosing parentheses.
ARGUMENTS: A string.
RETURNS: A pointer to the newly created multifield value.

### INTERNAL FUNCTIONS

#### InitializeFactHashTable

PURPOSE: Initializes the FactHashTable.

#### ResetFacts

PURPOSE: Resets the facts whenever a reset command is performed. This function is also used for the clear command.
Fact Commands Module

The Fact Commands Module (factcom.c) provides a number of commands for manipulating and examining facts. The commands provided are **assert**, **retract**, **save-facts**, **load-facts**, **facts**, **fact-index**, **dependencies**, **dependents**, **set-fact-duplication**, and **get-fact-duplication**.

**GLOBAL VARIABLES**

None.

**INTERNAL VARIABLES**

None.

**GLOBAL FUNCTIONS**

**InitFactCommands**

**PURPOSE:** Makes appropriate **DefineFunction** calls to notify CLIPS of functions defined in this module.

**INTERNAL FUNCTIONS**

**Fact Commands**

**PURPOSE:** A series of commands which define the fact commands listed above. See the **Basic Programming Guide** for more detail on individual functions.

**OTHER NOTES:** Some functionality for these commands is provided in other modules.
The Deffacts Manager Module (deffacts.c) manages all aspects of deffacts construct including parsing, execution, and removal. For a description of the deffacts construct, see the Basic Programming Guide. The deffacts construct capability can be removed by using the appropriate compile flag in the setup header file.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

| DeffactsArray | PURPOSE: A pointer to an array of deffacts loaded using the bload command. |
| DeletionsLegal | PURPOSE: A boolean flag indicating whether deffacts can be deleted (deffacts cannot be deleted while they are being reset). |
| LastDeffacts | PURPOSE: A pointer to the last deffacts in the ListOfDeffacts. |
| ListOfDeffacts | PURPOSE: A linked list of all the currently defined deffacts. |
| NumberOfDeffacts | PURPOSE: An integer count of the number of facts in the ListOfDeffacts. |

GLOBAL FUNCTIONS

| CreateInitialFactDeffacts | PURPOSE: Creates the initial-fact deffacts. |
| DeleteDeffacts | PURPOSE: Deletes a deffacts from the ListOfDeffacts. | ARGUMENTS: A pointer to the deffacts to be deleted. |
RETURNS: Boolean value. TRUE if the deffacts was found and deleted, otherwise FALSE.

### DeleteNamedDeffacts

**PURPOSE:** Deletes a named deffacts from the **ListOfDeffacts**.

**ARGUMENTS:** The name of the deffacts to be deleted.

**RETURNS:** Boolean value. TRUE if the deffacts was found and deleted, otherwise FALSE.

### FindDeffacts

**PURPOSE:** Finds a named deffacts in the **ListOfDeffacts**.

**ARGUMENTS:** The name of the deffacts to be found.

**RETURNS:** A pointer to the deffacts if found, otherwise NULL.

### GetDeffactsName

**PURPOSE:** Returns the name of a deffacts.

**ARGUMENTS:** A pointer to a deffacts.

**RETURNS:** String name of the deffacts.

### GetDeffactsPPForm

**PURPOSE:** Returns the pretty print representation of a deffacts.

**ARGUMENTS:** A pointer to a deffacts.

**RETURNS:** The string pretty print representation of the deffacts.

### GetNextDeffacts

**PURPOSE:** Allows access to the **ListOfDeffacts**.

**ARGUMENTS:** A pointer to a deffacts in the **ListOfDeffacts**.

**RETURNS:** If passed a NULL pointer, returns the first deffacts in the **ListOfDeffacts**. Otherwise, returns the next deffacts following the deffacts passed as an argument.
InitializeDeffacts

PURPOSE: Initializes the deffacts construct. Creates the initial-fact deffacts, adds reset, clear, save, bload, bsave, and constructs-to-c functions for deffacts, and defines the functions undeffacts, list-deffacts, and ppdeffacts.

IsDeffactsDeletable

PURPOSE: Indicates whether a deffacts can be deleted.

ARGUMENTS: A pointer to a deffacts.

RETURNS: Boolean value. TRUE if the deffacts can be deleted, otherwise FALSE.

ListDeffacts

PURPOSE: Displays the ListOfDeffacts.

ListDeffactsCommand

PURPOSE: Implements the list-deffacts command. Uses the driver function ListDeffacts.

PPDeffacts

PURPOSE: Pretty prints a deffacts.

ARGUMENTS: Name of deffacts to be pretty printed and logical name of the output source.

PpdeffactsCommand

PURPOSE: Implements the ppdeffacts command. Uses the driver function PPDeffacts.

RemoveAllDeffacts

PURPOSE: Removes all deffacts from the ListOfDeffacts.

SetListOfDeffacts

PURPOSE: Sets the ListOfDeffacts to the specified value. Normally used when initializing a run-time module or when bloading a binary file to install the ListOfDeffacts.

ARGUMENTS: A pointer to a linked list of deffacts.
**UndeffactsCommand**

**PURPOSE:** Implements the `undeffacts` command.

**INTERNAL FUNCTIONS**

**ClearDeffacts**

**PURPOSE:** Deffacts construct clear function. Removes all deffacts and creates the initial-fact deffacts.

**Deffacts Bload/Bsave Functions**

**PURPOSE:** A set of functions used by the `bload` and `bsave` commands to process the deffacts construct. These functions are made available to the `bload` and `bsave` commands by calling the function `AddBinaryItem`.

**Deffacts Constructs-To-C Functions**

**PURPOSE:** A set of functions used by the `constructs-to-c` command to process the deffacts construct. These functions are made available to the `constructs-to-c` command by calling the function `AddCodeGeneratorItem`.

**ParseDeffacts**

**PURPOSE:** Coordinates all actions necessary for the construction of a deffacts into the current environment. Called to parse a deffacts construct.

**ARGUMENTS:** Logical name from which deffacts input is read.

**OTHER NOTES:** Makes use of parsing functions from other modules such as the `GetConstructNameAndComment` function and the `BuildRHSAssert` function.

**ResetDeffacts**

**PURPOSE:** Deffacts construct reset function. Asserts all facts associated with deffacts into the `FactList`.

**SaveDeffacts**

**PURPOSE:** Deffacts construct save function. Pretty prints all deffacts to the given logical name.

**ARGUMENTS:** A logical name to which output is sent.
Defglobal Manager Module

The Defglobal Manager Module (defglobl.c) manages all aspects of defglobal construct including parsing, execution, and removal. For a description of the defglobal construct, see the Basic Programming Guide. The defglobal construct capability can be removed by using the appropriate compile flag in the setup header file.

GLOBAL VARIABLES

<table>
<thead>
<tr>
<th>ChangeToGlobals</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>

INTERNAL VARIABLES

<table>
<thead>
<tr>
<th>BDefglobalArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BDefglobalPointersArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DefglobalArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ListOfDefglobals</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NumberOfDefglobals</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>
**ResetGlobals**

**PURPOSE:** Boolean flag. If TRUE, indicates that globals will be reset to their original values when a `reset` command is performed. By being reset, the original expression associated with the global variable is reevaluated and then assigned to the global variable. If this flag is FALSE, then global variable values are not changed during a reset.

**SizeOfDefglobalArray**

**PURPOSE:** An integer count of the maximum number of global variables which can be stored in the `DefglobalArray`.

**WatchGlobals**

**PURPOSE:** Boolean flag. If TRUE, indicates that changes to globals should be displayed.

**GLOBAL FUNCTIONS**

**ClearDefglobals**

**PURPOSE:** Defglobals construct clear function. Removes all defglobals.

**FindDefglobal**

**PURPOSE:** Finds a named defglobal in the `DefglobalArray`.

**ARGUMENTS:** The name of defglobal to be found.

**RETURNS:** A pointer to the defglobal if found, otherwise NULL.

**GetActualDefglobal**

**PURPOSE:** Given a pointer returned by `FindDefglobal` or `GetNextDefglobal`, returns a pointer to the data structure where the global variable information is actually stored.

**ARGUMENTS:** A pointer to a `ListOfDefglobals` data structure which contains a pointer to a defglobal data structure.

**RETURNS:** A pointer to a defglobal data structure.

**GetDefglobalValue**

**PURPOSE:** Gets the value of a global variable.
ARGUMENTS: The name of the global variable and a pointer to a data structure in which the value of the global variable is to be stored.

RETURNS: Boolean value. TRUE if the global variable was found, otherwise FALSE.

GetDefglobalName

PURPOSE: Returns the name of a defglobal.

ARGUMENTS: A pointer to a defglobal.

RETURNS: String name of the defglobal.

GetDefglobalPPForm

PURPOSE: Returns the pretty print representation of a defglobal and its original expression value when it was defined.

ARGUMENTS: A pointer to a buffer in which to store the pretty print representation, the size of the buffer, and a pointer to a defglobal.

RETURNS: No return value. The buffer passed as an argument is used to store the pretty print representation.

GetDefglobalValueForm

PURPOSE: Returns the pretty print representation of a defglobal and its current value.

ARGUMENTS: A pointer to a buffer in which to store the pretty print representation, the size of the buffer, and a pointer to a defglobal.

RETURNS: No return value. The buffer passed as an argument is used to store the pretty print representation.

GetGlobalsChanged

PURPOSE: Returns the value of ChangeToGlobals.

GetIndexedDefglobal

PURPOSE: Given an integer index $n$, returns a pointer to the $n$th defglobal data structure in the DefglobalsArray.

ARGUMENTS: An integer index.
RETURNS: A pointer to a defglobal data structure.

---

**GetNextDefglobal**

PURPOSE: Allows access to the *ListOfDefglobals*.

ARGUMENTS: A pointer to a defglobal in the *ListOfDefglobals*.

RETURNS: If passed a NULL pointer, returns the first defglobal in the *ListOfDefglobals*. Otherwise, returns the next defglobal following the defglobal passed as an argument.

---

**GetNumberOfDefglobals**

PURPOSE: Returns the value of the *NumberOfDefglobals*.

RETURNS: An integer value.

---

**GetResetGlobals**

PURPOSE: Returns the current value of the *ResetGlobals* flag.

RETURNS: A boolean value.

---

**GetResetGlobalsCommand**

PURPOSE: Implements the *get-reset-globals* command.

---

**GlobalRtnUnknown**

PURPOSE: Access function placed within CLIPS expressions to retrieve the values of global variables.

ARGUMENTS: A pointer to a data structure in which to return a value. The integer index which indicates which global value to retrieve is stored in the argument list of this function’s expression.

RETURNS: No value. The value of the defglobal is stored in the data structure passed as an argument.

---

**InitializeDefglobal**

PURPOSE: Initializes the defglobal construct. Creates the globals watch item, adds *reset, clear, save, bload, bsave*, and *constructs-to-c* functions for defglobals, and defines the functions *set-reset-globals* and *get-reset-globals*. 

---
**ListDefglobals**

**PURPOSE:** Displays the **ListOfDefglobals** allow with their current values.

**ListDefglobalsCommand**

**PURPOSE:** Implements the **list-defglobals** command.

**PpdefglobalCommand**

**PURPOSE:** Implements the **ppdefglobal** command.

**QFindDefglobal**

**PURPOSE:** Finds a named defglobal in the **DefglobalArray**.

**ARGUMENTS:** The name of defglobal to be found. This argument is specified as a pointer to a **SymbolTable** entry rather than a character string.

**RETURNS:** A pointer to the defglobal if found, otherwise NULL.

**QGetDefglobalValue**

**PURPOSE:** Gets the value of a global variable.

**ARGUMENTS:** The integer index of the global variable and a pointer to a data structure in which the value of the global variable is to be stored.

**OTHER NOTES:** This function is quicker than **GetDefglobalValue** since the position of the global variable in the **DefglobalArray** does not have to be determined.

**QSetDefglobalValue**

**PURPOSE:** Sets the value of a global variable.

**ARGUMENTS:** The integer index of the global variable and a pointer to a data structure in which the new value of the global variable is stored.

**RETURNS:** Boolean value. TRUE if the global variable was found and its value changed, otherwise FALSE.

**OTHER NOTES:** This function is quicker than **SetDefglobalValue** since the position of the global variable in the **DefglobalArray** does not have to be determined.
ReplaceGlobalVariable

PURPOSE: Replaces a reference to a global variable within an expression with a function call to GlobalRtnUnknown that refers to the variable by an index for quick reference.

ARGUMENTS: A pointer to an expression.

RETURNS: Boolean value. TRUE if the global variable reference was replace, otherwise FALSE (the global could not be found).

ResetDefglobals

PURPOSE: Defglobals construct reset function. If the ResetGlobals flag is TRUE, then all global variables are reset to their original values.

SetDefglobalValue

PURPOSE: Sets the value of a global variable.

ARGUMENTS: The name of the global variable and a pointer to a data structure in which the new value of the global variable is stored.

RETURNS: Boolean value. TRUE if the global variable was found and its value changed, otherwise FALSE.

SetGlobalsChanged

PURPOSE: Sets value of ChangeToGlobals.

ARGUMENTS: Boolean value.

SetListOfDefglobals

PURPOSE: Sets the ListOfDefglobals, DefglobalArray, and NumberOfDefglobals, and SizeOfDefglobalArray to the specified values. Normally used when initializing a run-time module or when bloadding a binary file.

ARGUMENTS: A pointer to a linked list of defglobals, an array in which the defglobal values are stored, and the number of defglobals contained in the array (which is also the size of the array).

SetResetGlobals

PURPOSE: Sets the current value of the ResetGlobals flag.
ARGUMENTS: A boolean value (the new value of the flag).
RETURNS: A boolean value (the old value of the flag).

### SetResetGlobalsCommand

**PURPOSE:** Implements the `set-reset-globals` command.

### INTERNAL FUNCTIONS

#### AddDefglobal

**PURPOSE:** Adds a global variable to the `ListOfDefglobals` and the `DefglobalArray`. If the global variable already exists, then it is replaced.

**ARGUMENTS:** The name of the global variable, a pointer to a data structure in which the global's initial value is stored, and a pointer to the expression to be evaluated to determine the global's value whenever it is reset.

**OTHER NOTES:** The `DefglobalArray` is dynamically expanded if the `SizeOfglobalArray` is not large enough to contain the new global variable.

#### Defglobal Bload/Bsave Functions

**PURPOSE:** A set of functions used by the `bload` and `bsave` commands to process the defglobal construct. These functions are made available to the `bload` and `bsave` commands by calling the function `AddBinaryItem`.

#### Defglobal Constructs-To-C Functions

**PURPOSE:** A set of functions used by the `constructs-to-c` command to process the defglobal construct. These functions are made available to the `constructs-to-c` command by calling the function `AddCodeGeneratorItem`.

#### GetVariableDefinition

**PURPOSE:** Parses a single variable definition within a defglobal construct. If no errors occur while defining the variable, the function `AddDefglobal` is called to add the new global variable to the `ListOfDefglobals`.

**ARGUMENTS:** Logical name from which defglobal input is read and a pointer to an integer error flag.
RETURNS: Boolean value. FALSE if an error occurred while parsing, otherwise TRUE. The value of the error flag passed as an argument is also set by this function.

OTHER NOTES: Uses the function ParseAtomOrExpression to parse the expression assigned to the global variable. The function EvaluationExpression is then called to determine the initial value of the variable.

**ParseDefglobal**

PURPOSE: Coordinates all actions necessary for the construction of a defglobal into the current environment. Called to parse a defglobal construct.

ARGUMENTS: Logical name from which defglobal input is read.

RETURNS: Boolean value. TRUE if an error occurred while parsing, otherwise FALSE.

OTHER NOTES: Uses the function GetVariableDefinition to perform the majority of parsing.

**SaveDefglobals**

PURPOSE: Defglobal construct save function. Pretty prints all defglobals to the given logical name.

ARGUMENTS: A logical name to send output.
The Defrule Parser Module (ruleprsr.c) coordinates the parsing of the LHS of the rule (as well as providing functions for parsing the RHS of a rule). LHS conditional elements are represented internally using the following format:

1st Conditional Element --> CE information

2nd Conditional Element --> CE information

3rd Conditional Element --> CE information

•

nth Conditional Element --> CE information

If the conditional element is a test CE, the CE information will be an expression stored using the standard format for an expression. The CE information for a connected conditional element (an and CE, or CE, or logical CE) follows the format shown above. The information for a pattern CE or a not CE is used to represent the fields of the pattern.

As an examples, the conditional elements for the following rule

(defrule example
  (pattern 1)
  (or (pattern 2a)
      (pattern 2b))
  (not (pattern 3))
  (pattern 4)
=>)

would be stored as
pattern CE --> pattern 1 information

or CE --> pattern CE --> pattern 2a information

not CE --> pattern 2b information

not CE --> pattern 3 information

pattern CE --> pattern 4 information

The CE information for pattern CEs and not CEs is stored using the following format:

1st field --> 2nd field --> 3rd field ... nth field

field info  field info  field info  field info

Information for each field is stored in the following format:

first variable (if any)

1st | connective constraint --> & connective constraints

2nd | connective constraint --> & connective constraints

... 

nth | connective constraint --> & connective constraints

The first-binding occurrence of a variable is stored first in the structure (if it exists). A first-binding occurrence of a variable for a field in a pattern is a variable by itself or a variable followed by an & connective constraint. The variable cannot be negated. First occurrences of the variable ?x in a field of a pattern would include

?x

?x&blue|green
but not

\[
\sim ?x
\]

\[
?x \mid ?y
\]

\[
\text{red} \& ?x
\]

The structure to contain the first binding variable is also used to indicate whether
the field should match a single field value or a multifield value. Fields without a binding
variable are considered to match against a single field value.

The subsequent bottom links connected to the binding variable structure contain
information about the list of \mid connective constraints found within the field. Each \mid
connective constraint of a field can be accessed through the bottom link of the
structure. The first structure to the immediate right of each \mid connective constraint
represents the first constraint associated with the \mid connective constraint. Structures to
the right of this first constraint represent other constraints associated with the \mid
connective constraint through the use of the \& connective constraint. Individual
constraints can be literal constraints, predicate constraints, return value constraints,
and/or variable constraints. Any of these constraints may be negated using the \sim
connective constraint. When grouping constraints, the \mid connective constraint in a field
constraint is given a lower precedence than the \& connective constraint.

For example, the following field found in a pattern

\[
?x \& \text{:numberp(?x)} \& (=?y 3) \mid \text{:wordp(?x)} \& \sim \text{red} \& \sim \text{green}
\]

would be represented as

![Diagram]

| single field variable x |
|---|---|
| numberp(?x) \rightarrow (=?y 3) |
| wordp(?x) \rightarrow \sim\text{red} \rightarrow \sim\text{green} |

### GLOBAL VARIABLES

<table>
<thead>
<tr>
<th>GlobalSalience</th>
</tr>
</thead>
</table>

**PURPOSE:** An integer used to store the evaluated value of the salience
when the rule is defined (i.e. the evaluated value of the variable SalienceExpression when the rule is defined).

<table>
<thead>
<tr>
<th>SalienceExpression</th>
</tr>
</thead>
</table>

**PURPOSE:** A pointer to the expression used in the salience declaration
of a rule (which may either be a constant integer, global
variable, or a function call).
INTERNAL VARIABLES

<table>
<thead>
<tr>
<th>LHSError</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: A global boolean value used to indicate whether an error has occurred in one of the rule parsing routines.</td>
</tr>
</tbody>
</table>

GLOBAL FUNCTIONS

<table>
<thead>
<tr>
<th>ParseRuleLHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Coordinates all the actions necessary for parsing the LHS of a rule including the reordering of pattern conditional elements to conform with the CLIPS Rete topology.</td>
</tr>
<tr>
<td>ARGUMENTS: Logical name from which rule input is read and a pointer to a token structure in which scanned tokens are placed.</td>
</tr>
<tr>
<td>RETURNS: A pointer to a linked structure containing the intermediate LHS representation of a rule. If an error has occurred during parsing, a null pointer is returned.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ParseRuleRHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Coordinates all the actions necessary for parsing the RHS of a rule.</td>
</tr>
<tr>
<td>ARGUMENTS: Logical name from which rule input is read.</td>
</tr>
<tr>
<td>RETURNS: An expression structure representing the RHS of a rule.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RestrictionParse</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Parses a single field within a pattern. This field may either be a single field wildcard, a multifield wildcard, a single field variable, a multifield variable, or a series of connected constraints.</td>
</tr>
<tr>
<td>ARGUMENTS: Logical name from which input is read and a pointer to a token structure.</td>
</tr>
<tr>
<td>RETURNS: Intermediate LHS representation of the field.</td>
</tr>
</tbody>
</table>
INTERNAL FUNCTIONS

AssignmentParse

PURPOSE: Finishes the parsing of pattern conditional elements that have been bound to a variable.

ARGUMENTS: Logical name from which input is read, and name of the variable (or the fact address) to which the pattern CE is bound.

RETURNS: Intermediate LHS representation of the assigned pattern conditional element.

ConjunctiveRestrictionParse

PURPOSE: Parses a single constraint field in a pattern that is not a single field wildcard, multifield wildcard, or multifield variable. The field may consist of a number of subfields tied together using the & connective constraint and/or the | connective constraint.

ARGUMENTS: Logical name from which input is read, and a pointer to a token structure.

RETURNS: Intermediate LHS representation of the field.

ConnectedPatternParse

PURPOSE: Handles parsing of the connected conditional elements (i.e. those conditional elements that may contain one or more other conditional elements). The connected conditional elements include the and CE, the or CE, and the logical CE.

ARGUMENTS: Logical name from which input is read, and a pointer to a token structure.

RETURNS: Intermediate LHS representation of the connected conditional element.

CreateInitialPattern

PURPOSE: Creates an LHS representation of the pattern (initial-fact) for rules which do not contain an LHS.

RETURNS: Intermediate LHS representation of the pattern (initial-fact).
## DeclarationParse

**PURPOSE:** Parses a defrule declaration. Only salience declarations are currently allowed.

**ARGUMENTS:** Logical name from which input is read.

**RETURNS:** Nothing. Sets value of the variables `GlobalSalience` and `SalienceExpression`.

## GroupPatterns

**PURPOSE:** Groups a series of connected conditional elements together.

**ARGUMENTS:** Logical name from which input is read, type of token which terminates the CE grouping, and string representation of the terminating token.

**RETURNS:** Intermediate LHS representation of the grouped patterns.

## LHSPattern

**PURPOSE:** Parses a single conditional element found on the LHS of a rule. Conditional element types include pattern CEs (which may be assigned to a variable), test CEs, not CEs, logical CEs, and CEs, and or CEs.

**ARGUMENTS:** Logical name from which input is read, and the type of token which terminates the conditional element grouping in which the conditional element is found (e.g. a pattern CE parsed within an and CE is terminated by a parenthesis while a pattern CE not enclosed by another CE is terminated by the => symbol).

**RETURNS:** Intermediate LHS representation of the LHS conditional element.

## LiteralRestrictionParse

**PURPOSE:** Parses a subfield of a field. The subfield may be a literal constraint, a predicate constraint, a return value constraint, or a variable constraint. The constraints may also be negated using the ~ connective constraint.

**ARGUMENTS:** Logical name from which input is read, and a pointer to a token structure.

**RETURNS:** Intermediate LHS representation of the subfield.
**NotPatternParse**

**PURPOSE:** Handles parsing of not conditional elements.

**ARGUMENTS:** Logical name from which input is read.

**RETURNS:** Intermediate LHS representation of the not conditional element.

**RuleBodyParse**

**PURPOSE:** Parses the LHS of a rule, but does not reorder any of the LHS patterns to conform with the CLIPS Rete Topology.

**ARGUMENTS:** Logical name from which rule input is read and a pointer to a token structure in which scanned tokens are placed.

**RETURNS:** A pointer to a linked structure containing the intermediate LHS representation of a rule. If an error has occurred during parsing, a null pointer is returned.

**SequenceRestrictionParse**

**PURPOSE:** Parses a sequence of constraint fields found within a pattern. This function recognizes deftemplate patterns and will call the appropriate routines to parse these types of patterns.

**ARGUMENTS:** Logical name from which input is read, and a pointer to a token structure.

**RETURNS:** Intermediate LHS representation of the sequence of fields.

**SimplePatternParse**

**PURPOSE:** Parses a simple pattern (an opening parenthesis followed by one or more fields followed by a closing parenthesis).

**ARGUMENTS:** Logical name from which input is read, and a pointer to a token structure.

**RETURNS:** Intermediate LHS representation of the simple pattern conditional element.

**TagLHSLogicalNodes**

**PURPOSE:** Marks all and and or conditional elements contained within a logical conditional element as having the properties associated with a logical CE.
ARGUMENTS: The LHS representation of a logical conditional element.

TestPattern

PURPOSE: Handles parsing of test conditional elements.

ARGUMENTS: Logical name from which input is read.

RETURNS: Intermediate LHS representation of the test conditional element.
Basic Rete topology only allows a pattern conditional element to stand by itself or to be modified with the \textbf{not} conditional element. In addition, the LHS is enclosed within an implied \textbf{and} conditional element. Combinations of \textbf{and} conditional elements and \textbf{or} conditional elements are not allowed using basic Rete topology. CLIPS allows these conditional elements to be used in combination by generating multiple rules which conform to basic Rete topology from single instances of rules which do not conform to basic Rete topology. The Reorder Module (reorder.c) reorders a single LHS which may or may not conform to basic Rete topology into one or more LHSs which do conform to basic Rete topology. Reordered LHSs have a single top-level \textbf{or} pattern conditional element (with each argument of the \textbf{or} conditional element representing a separate rule which must be generated) with multiple \textbf{and} conditional elements containing one or more pattern conditional elements or \textbf{not} conditional elements but no other types of conditional elements. For the purposes of reordering, the \textbf{logical} conditional element behaves identically to the \textbf{and} conditional element.

Reordering involves two major steps: transformation and reduction. Transformation involves changing a conditional element from one form to another equivalent form. The transformation performed when reordering patterns involves replacing \textbf{and/or} conditional elements with equivalent \textbf{or/and} conditional elements. For example,

\begin{verbatim}
(and (or (a) (b))
(or (c) (d)))
\end{verbatim}

can be replaced with

\begin{verbatim}
(or (and (a)
(or (c) (d)))
(and (b)
(or (c) (d))))
\end{verbatim}

This transformation makes use of the observation that the conditional elements contained within (or (a) (b)) can be extracted and combined individually with an \textbf{and} conditional element with copies of the (or (c) (d)) conditional element. The resulting set of conditional elements can then be placed together using an \textbf{or} conditional element. This transformation stated more generally is

\begin{verbatim}
(and (<CE-a-1> ... (or <CE-o-1> ... <CE-o-n>) ...) <CE-a-n>)
\end{verbatim}

can be replaced with

\begin{verbatim}
(or (and <CE-a-1> ... <CE-o-1> ... (pattern an)) ...) (and <CE-a-1> ... <CE-o-n> ... (pattern an)))
\end{verbatim}

Reduction involves simplifying conditional elements. The reduction used when reordering conditional elements involves removing redundant information. For example, a CE such as (and (and <CE-1> <CE-2>) can be simplified to (and <CE-1>
This type of reduction will be referred to as adjacency reduction. As another example,

\[(\text{or} \ (\text{and} \ (\text{and} \ (a) \ (b)) \ (\text{and} \ (c) \ (d))))\]

can be converted to

\[(\text{or} \ (\text{and} \ (a) \ (b)) \ (\text{and} \ (c) \ (d)))\]

Note that, for this type of simplification, the **and/or** conditional elements must be adjacent. For example, the following CE would not be simplified by this type of reduction:

\[(\text{or} \ (\text{and} \ (\text{or} \ (a))))\]

As a point of interest, advanced Rete topology allows the **and** conditional element to be incorporated directly into the Rete Join Network. This feature, called joins from the right, is discussed in further detail in an article by IBM. Currently, joins from the right are not implemented in CLIPS but, if added, would require changes to the manner in which the reordering of conditional elements is accomplished.

### GLOBAL VARIABLES
None.

### INTERNAL VARIABLES
None.

### GLOBAL FUNCTIONS

<table>
<thead>
<tr>
<th>CopyNodes</th>
</tr>
</thead>
</table>
| **PURPOSE:** Copies a set of patterns.  
**ARGUMENTS:** Patterns to be copied.  
**RETURNS:** A copy of the patterns. |

<table>
<thead>
<tr>
<th>GetNode</th>
</tr>
</thead>
</table>
| **PURPOSE:** Creates an empty node structure used for building patterns.  
**RETURNS:** A pointer to an empty node structure initialized with default values. |

<table>
<thead>
<tr>
<th>ReorderPatterns</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PURPOSE:</strong> Reorders a group of patterns to accommodate CLIPS Rete topology.</td>
</tr>
</tbody>
</table>
ARGUMENTS: A group of patterns.

RETURNS: A modified group of patterns that contains a single top-level or conditional element followed by one or more and conditional elements.

---

**ReturnNodes**

PURPOSE: Returns a set of patterns to the free pool of memory.

ARGUMENTS: Patterns to be returned.

---

**INTERNAL FUNCTIONS**

---

**AdjacentReduction**

PURPOSE: Performs adjacency reduction on a group of patterns.

ARGUMENTS: A group of patterns.

RETURNS: A modified group of patterns.

---

**ReverseOR**

PURPOSE: Performs a transformation on logical and/or pattern operators to change them to logical or/and pattern operators.

ARGUMENTS: A group of patterns.

RETURNS: A modified group of patterns.
Variable Manager Module

The Variable Manager Module (variable.c) provides a set of access functions which are used to retrieve the results of the analysis of the LHS of a rule. Some of the functions provided can be used to determine the location of a variable on the LHS of a rule and to obtain the expressions generated for the pattern and join networks. These access functions are utilized by the Build Module when it adds a rule to the rule network.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

<table>
<thead>
<tr>
<th>AnalysisExpressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Maintains a list for each pattern of the expressions to be evaluated in the pattern and join networks.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CurrentPatternInfo</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Maintains a list of information about patterns and the variables contained within them.</td>
</tr>
</tbody>
</table>

GLOBAL FUNCTIONS

<table>
<thead>
<tr>
<th>CountJoins</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Determines number of joins needed for the LHS of a rule</td>
</tr>
<tr>
<td>RETURNS: An integer representing the number of joins needed for the LHS of a rule (in essence, the number of patterns on the LHS of the rule).</td>
</tr>
<tr>
<td>OTHER NOTES: Accesses AnalysisExpressions to derive the information.</td>
</tr>
<tr>
<td>OTHER NOTES: Accesses AnalysisExpressions to derive the information.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CountPatternFields</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Determines the number of fields in a pattern.</td>
</tr>
<tr>
<td>ARGUMENTS: Pattern number.</td>
</tr>
<tr>
<td>RETURNS: An integer representing number of fields in the pattern.</td>
</tr>
</tbody>
</table>
OTHER NOTES: Accesses the variable **AnalysisExpressions** to derive the information.

---

**ExpressionComplexity**

**PURPOSE:** Determines the complexity of an expression for use with the lex, mea, simplicity, and complexity conflict resolution strategies.

**ARGUMENTS:** An pointer to an expression.

**RETURNS:** An integer value representing the complexity of the expression. Each function call contained within an expression adds one to the complexity of the expression (with an initial complexity of zero). Calls to the **and**, **or**, and **not** functions do not increase the complexity of an expression, but function calls made within these functions do.

---

**FindVariable**

**PURPOSE:** Searches for the location of the first binding occurrence of a variable on the LHS of a rule available to a specific pattern or expression. Such a variable must occur before the pattern and field in which the reference is made (with the exception of variables in deftemplate patterns for which forward referencing is allowed since the position of the fields in the pattern will be rearranged).

**ARGUMENTS:** Name of the variable being sought, first pattern in which to begin looking for the variable, current pattern and field with which the variable is associated (with a field value of -1 indicating that forward references are allowed), and a value indicating whether the variable is “inside” or “outside” the pattern (i.e. a variable inside a **not** conditional element can refer to other variables within that CE, but variables outside of a **not** CE cannot refer to variables within a **not** CE).

**RETURNS:** A pointer to a variable information structure if the variable is found; otherwise NULL.

**OTHER NOTES:** Starting pattern and inside/outside values are very useful. By setting the starting pattern to the current pattern, it can be determined whether a variable reference within a pattern can be compared to another variable within the same pattern as opposed to using a variable in a previous pattern. This information is useful when determining which expressions can be placed in the pattern network. The inside/ outside value allows strict scoping of variables within **not** CE's. For
example, a test CE following a not CE is outside of the not CE and may not reference any of the variables within the not CE, while an expression associated with a predicate constraint used within the not CE is inside the not CE and may reference variables used within the not CE. If the field index is set to -1, forward references of variables within a pattern will be allowed. This is allowed for template patterns since the variables may be referenced in the proper order within the original pattern but might be rearranged in an improper order when the actual pattern to be used is generated.

**FlushAnalysisExpressions**

**PURPOSE:** Returns structures associated with the global variable *AnalysisExpressions* and sets the variable to NULL.

**FlushVariableAnalysis**

**PURPOSE:** Purges all current information about patterns and variables.

**GetFactAddressPosition**

**PURPOSE:** Returns the pattern number (ranging from one to the number of patterns) corresponding to a fact address variable.

**ARGUMENTS:** Name of the fact address variable.

**RETURNS:** Pattern position to which the fact address variable is bound (or zero if not found).

**GetJoinLogic**

**PURPOSE:** Returns RHS join logic for a particular pattern.

**ARGUMENTS:** Pattern number.

**RETURNS:** A character. The character ‘-’ is returned if the connected pattern is within a not CE and ‘+’ is returned if the connected pattern is not within a not CE. A ‘?’ is returned if the pattern number does not correspond to an analyzed join.

**GetNodeType**

**PURPOSE:** Returns pattern network logic for a field of a given pattern.

**ARGUMENTS:** Pattern and field number.
RETURNS: Logic type. If it can be found, it will be SINGLE, MULTIFIELD, or STOP. If it cannot be found, it will return UNKNOWN to signal an error.

OTHER NOTES: Accesses the variable AnalysisExpressions to derive the information.

---

**GetNotJoinExpression**

PURPOSE: Returns secondary join network expression for a particular pattern. The secondary expression is needed when test expressions are used after a not conditional element.

ARGUMENTS: Pattern number.

RETURNS: Secondary join network expression.

OTHER NOTES: Returns original copy of the expression and sets pointer to the expression to null. Hence, subsequent calls with the same pattern number will return null. Accesses AnalysisExpressions to derive the information.

---

**GetPatternExpression**

PURPOSE: Returns pattern network test for a particular pattern and field.

ARGUMENTS: Pattern and field number.

RETURNS: Pattern network expression.

OTHER NOTES: Returns original copy of the expression and sets pointer to the expression to NULL. Hence, subsequent calls with the same arguments will return NULL. Accesses AnalysisExpressions to derive the information.

---

**GetPrimaryJoinExpression**

PURPOSE: Returns the primary join network expression for a particular pattern.

ARGUMENTS: Pattern number.

RETURNS: Primary join network expression.

OTHER NOTES: Returns original copy of the expression and sets pointer to the expression to NULL. Hence, subsequent calls with the same pattern number will return NULL. Accesses AnalysisExpressions to derive the information.
GetRelationForPattern

PURPOSE: Returns the relation name (if any) associated with the first field of an LHS pattern.

ARGUMENTS: An integer index representing the pattern to be checked.

RETURNS: A pointer to the relation name symbol if it exists; otherwise NULL.

GetVariableInformation

PURPOSE: Returns the value of the variable CurrentPatternInfo.

RETURNS: The variable CurrentPatternInfo.

PatternHasTemplate

PURPOSE: Determines whether a pattern on the LHS of a rule has an associated deftemplate.

ARGUMENTS: An integer index representing the pattern to be checked.

RETURNS: Boolean value. True if the first field of the LHS pattern is associated with a deftemplate; otherwise false.

RuleComplexity

PURPOSE: Determines the complexity of a rule for use with the lex, mea, simplicity, and complexity conflict resolution strategies.

ARGUMENTS: None. The complexity is computed for the rule being currently analyzed. The variable AnalysisExpressions is used to derive the complexity information.

RETURNS: An integer value representing the complexity of the rule. The rule complexity is the sum of the complexity of each expression associated with the join or pattern network for the rule computed using the function ExpressionComplexity.

SetRuleInformation

PURPOSE: Sets the value of the variable AnalysisExpressions.

ARGUMENTS: The new value.

SetVariableInformation

PURPOSE: Sets the value of the variable CurrentPatternInfo.
ARGUMENTS: The new value.

INTERNAL FUNCTIONS

None.
The Analysis Module (analysis.c) creates the appropriate function calls to be embedded in the join and pattern network. It also uses both the Variable Module and the Build Module to create expressions to be placed in the network. When the LHS representation of a rule is passed to the rule analysis function, several steps in the generation of an expression occur.

First, the Analysis Module determines the location of variables within the patterns and if any semantic errors involving the use of variables have occurred. It analyzes the set of LHS patterns to determine where variables are being bound. It keeps track of fact address variables the patterns to which they are bound and detects errors in the usage of variables.

Each pattern has the following information stored about it: Which pattern is it (first, second, third)? Is the pattern bound to a fact address variable; and, if so, what is the name of the variable? Is the pattern logically negated? Which variables are bound in this pattern?

Bound variables are variables which either stand alone in a field or are the first subfield of a field and are immediately followed by an & connective constraint. Bound variables have the following information stored about them: the variable name, the pattern and field numbers in which they are found, and whether the variable is a single- or multifield variable.

The typical error detected by the Analysis Module is a reference to a variable before it has been bound. The following rules all incorrectly reference the variable ?x.

(defrule error1
  (fact ~?x)
  =>)

(defrule error2
  (not (fact ?x))
  (data ?x)
  =>)

(defrule error3
  (data ?y)
  (test (> ?x ?y))
  =>)

(defrule error4
  (not (fact ?x))
  (test (> ?x 4))
  =>)

(defrule error5
  (data ?x | all)
  =>)

Rules error1 and error3 simply make a reference to the variable x before the variable has been bound. Rules error2 and error4 demonstrate that the scope of a variable first bound within a not CE is limited strictly to within the not CE. Rule error4 can be corrected by placing the test within the not CE using &:(> ?x 4). Rule error5
also makes an unbound reference to ?x. Variable ?x is the first variable in the field; however, it is connected with a \| connective constraint and, therefore, is not considered to be a binding occurrence of the variable.

Note that the deftemplate construct generates *normal* LHS patterns from the LHS template patterns used in a rule. Because fields may be listed in any order in a template pattern, it is possible for a converted template pattern to access a variable before that variable is defined. For example, given the following deftemplate,

```lisp
(deftemplate temp
  (field x)
  (field y))
```

the following two rules properly use the template patterns:

```lisp
(defrule example-1
  (temp (x ?x) (y ?y&~?x))
  =>)

(defrule example-2
  (temp (y ?y) (x ?x&~?y))
  =>)
```

Notice that, in the template patterns of both rules, variables are defined before they are used. However, when the conversion from LHS template patterns to normal LHS patterns is performed, the rules appear as follows:

```lisp
(defrule example-1
  (temp ?x ?y&~?x)
  =>)

(defrule example-2
  (temp ?x&~?y ?y)
  =>)
```

Rule example-1 has no forward references to variables; however, rule example-2 references the variable ?y before it is defined. Because fields in a template pattern may be specified in any order and the specified fields may be rearranged in generating the actual LHS pattern to be used, forward references to variables in a template pattern are allowed so long as the variable is contained somewhere within the pattern in which it is referenced first.

Once the variables within the patterns have been identified, the Generate Module can then be used to generate expressions for the pattern and join networks. Many factors must be considered when generating expressions for evaluation in the networks. Several examples bear mentioning.

```lisp
(defrule example1
  (foo ?x)
  (not (bar ?x))
  (test (> ?x 4))
  =>)
```
Rule example1 demonstrates that two separate expressions can be needed for joins with a not CE. The first expression needed for the not CE is performed on the pattern itself. This expression checks to see if the ?x in the bar fact is the same as the ?x in the foo fact. The expression (> ?x 4) references ?x in the foo fact but should not be associated with the other expression. This is necessary for the case where no bar facts exist. The expression comparing ?x in foo to ?x in bar does not have to be performed in this case. If the (> ?x 4) expression was associated with the other expression, the existence of any foo fact along with no bar facts would cause the rule to be activated. The rule should be activated only for foo's with ?x greater than 4.

(defun example2
  (foo ?x)
  (bar ?x ?x)
=>)

Rule example2 has two expressions which must be performed for the second pattern. The first expression ensures that the ?x in the bar fact is the same as the ?x in the foo fact. The second expression ensures that the ?x in the second field of the bar fact is the same as the ?x in the third field of the bar fact. The expression comparing across patterns must be done in the join network. The expression comparing ?x’s within the bar pattern can be performed in the pattern network, however.

(defun example3
  (foo ?x)
  (bar ?x | all)
=>)

Rule example3 demonstrates an example of an expression that must be moved from the pattern network to the join network. Because the second field in the bar fact has a comparison to a value first bound in another pattern, the expression for this field must be moved into the join network. An expression for the constant all cannot be performed in the pattern network because the element can either bind to all or to some as of yet unspecified value ?x.

(defun example4
  (foo ?x&:(numberp ?x))
=>)

Rule example4 is another example of an expression that can be evaluated in the pattern network since all arguments of numberp can be accessed in the pattern.

(defun example5
  (bar ?y)
  (foo ?x&:(> ?x ?y))
=>)

Rule example5 shows a predicate constraint which must be evaluated in the join network because of the reference to ?y bound outside of the pattern.

The Analysis Module generates a pattern network expression for every field in a pattern and a join network expression for every pattern. Not CEs may also have an additional join network expression. The Analysis Module determines which
expressions are performed in the pattern network and which are done in the join network. When possible, expressions should be evaluated in the pattern network.

If a particular field has no | connective constraints, few restrictions apply to expressions which can be evaluated in the pattern network. All tests for constants can be evaluated in the pattern network. Predicate constraints and return value constraints can be evaluated in the pattern network as long as references to variables in other patterns are not made. Expressions comparing two references of the same variable can be evaluated in the pattern network if both references are found in the same pattern and one reference is to a bound variable. All other expressions that reference variables outside of the pattern must be made in the join network. Note that test CE tests are always performed in the join network whereas predicate constraints and return value constraints may be performed either in the pattern or join network depending on the variables referenced.

If a field has an | connective constraint in it and references are made to a variable bound in another pattern that is not bound in this pattern, all expressions for this field must be performed in the join network. Rule example3 is an example of this type of reference.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

<table>
<thead>
<tr>
<th>DeftemplatePattern</th>
</tr>
</thead>
</table>

PURPOSE: Used to indicate whether a pattern being analyzed has an associated deftemplate.

GLOBAL FUNCTIONS

<table>
<thead>
<tr>
<th>CheckExpression</th>
</tr>
</thead>
</table>

PURPOSE: Verifies that variables within an expression have been referenced properly. All variables within an expression must have been previously defined.

ARGUMENTS: A pointer to the expression, the first pattern that can be checked for a variable reference, the current pattern and element with which the expression is associated, and a value indicating whether the expression is “inside” or “outside” the pattern.

RETURNS: If no error is detected, a pointer to the expression; otherwise null.
**CheckVariables**

**PURPOSE:** Verifies the proper use of variables on the LHS of a rule. Checks that fact addresses are not reused or used as variables within patterns and that variables within patterns are referenced properly.

**ARGUMENTS:** The LHS representation of the patterns (which contains no embedded **and CE**s or **or CE**s).

**RETURNS:** Boolean value. TRUE if an error is detected; otherwise FALSE.

**LogicalAnalysis**

**PURPOSE:** Analyzes for the correct use of **logical CE**s on the LHS of a rule. Gaps may not exist between **logical CE**s and **logical CE**s must occurs before other CEs on the LHS of the rule.

**ARGUMENTS:** A pointer to the intermediate LHS representation of a rule (which contains no embedded **and CE**s or **or CE**s).

**RETURNS:** -1 if an error was detected, otherwise the integer index of last **logical CE** on the LHS of the rule (ranging from one to the number of patterns in the rule). If the rule has no **logical CE**s, then zero is returned.

**RuleAnalysis**

**PURPOSE:** Analyzes a set of patterns for variable bindings, performs semantic error checking on the use of variables, and determines expressions which will be evaluated in the pattern and join networks.

**ARGUMENTS:** A pointer to the intermediate LHS representation of a rule.

**RETURNS:** Boolean value. TRUE if a semantic error occurred; otherwise FALSE.

**VariableAnalysis**

**PURPOSE:** Analyzes a set of patterns to determine the position of each pattern in the rule, whether the pattern is contained within a **not CE**, and if the pattern is bound to a fact address.

**ARGUMENTS:** The LHS representation of the patterns (which contains no embedded **and CE**s or **or CE**s).
OTHER NOTES: Creates the data structures and then calls the function SetVariableInformation.

INTERNAL FUNCTIONS

**AllVariablesInPattern**

**PURPOSE:** Determines if all variable references made in a field can be found within the containing pattern in previous fields. This is important for determining whether certain expressions can be evaluated in the pattern network as opposed to the join network.

**ARGUMENTS:** A pointer to the field and the pattern and field integer index numbers.

**RETURNS:** A boolean value. TRUE if all variable references are contained within the pattern; otherwise FALSE.

**BuildNetworkExpressions**

**PURPOSE:** Constructs an entry for each pattern CE and test CE associated with that pattern CE in the LHS of a rule. The entry contains information about pattern network expressions associated with each field and primary and secondary join expressions associated with the pattern.

**ARGUMENTS:** A pointer to the intermediate LHS representation of a rule.

**OTHER NOTES:** Creates the data structures and then calls the function SetRuleInformation.

**CheckFactAddress**

**PURPOSE:** Verifies that a fact address has not been used more than once and has not been used as a variable name.

**ARGUMENTS:** Name of fact address variable and pattern index to which it is bound.

**RETURNS:** Boolean value. TRUE if an error is detected; otherwise FALSE.

**CheckPattern**

**PURPOSE:** Verifies that variables within a pattern have been referenced properly (i.e. that variables have been previously bound if they are not a binding occurrence).
ARGUMENTS: The LHS representation of the pattern and the pattern index of the pattern.

RETURNS: Boolean value. TRUE if an error is detected; otherwise FALSE.

**ExtractAnds**

PURPOSE: Loops through a single set of subfields bound together by an & connective constraint in a field and generates expressions needed for testing conditions in the pattern and join network.

ARGUMENTS: A pointer to the intermediate LHS representation of the subfields connected by the & connective constraint, the integer index of the pattern and field in which the subfields occur, a boolean flag indicating whether certain tests may be performed in the pattern network, and a pointer to a data structure in which expressions to be used in the pattern and join network will be returned.

RETURNS: No formal return parameter. Returns expressions to be evaluated in the pattern network and expressions to be evaluated in the join network in a data structure passed as an argument.

OTHER NOTES: Uses Generate Module to build subfield expressions.

**FieldConversion**

PURPOSE: Generates expressions to be evaluated in the pattern network and join network for a field in a pattern. Uses function ExtractAnds to generate subfield expressions, then combines the subfield expressions together.

ARGUMENTS: A pointer to the intermediate LHS representation of the pattern field, the integer index of the pattern and field in the LHS of the rule, and a pointer to a data structure in which expressions to be used in the pattern and join network will be returned.

RETURNS: No formal return parameter. Returns expressions to be evaluated in the pattern network and expressions to be evaluated in the join network in a data structure passed as an argument.

**GetVariables**

PURPOSE: Extracts the variable references from a single pattern.
ARGUMENTS: Intermediate LHS representation of the pattern and the pattern index number (e.g., first, second, or third pattern in the rule).

RETURNS: A linked list of structures containing information about each variable in the pattern.
The Generate Module (generate.c) transforms the basic syntax primitives of a pattern into expressions which will be placed in the pattern and join networks. & and | connectives are respectively replaced with the equivalent function call to the **and** function or the **or** function. Other primitives bear mentioning as to the type of replacements that are made.

Access to specific variables from the join network or RHS uses the **getvar** function.

```
(getvar <pattern-m> <field-n>)
```

Access to specific variables from the pattern network uses the **getfield** function, which only requires a field specification since the specific pattern is implied by the current fact.

```
(getfield <field-m>)
```

Comparison of variables in the join network uses **eqvar** and **neqvar** to test, respectively, if a set of variables is either equal or not equal. The pattern associated with the first field in the comparison is assumed to be the pattern entering from the RHS of the join in which the expression is located. The depth field of the join structure is used to determine this pattern index.

```
(eqvar field-n pattern-x field-y)
(neqvar field-n pattern-x field-y)
```

Equivalent functions for the pattern network are **eqfield** and **neqfield**.

```
(eqfield field-m field-n)
(neqfield field-m field-n)
```

Constants are evaluated in the pattern network using the **constant** and **notconstant** functions.

```
(constant <value>)
(notconstant <value>)
```

Constants are evaluated in the join network using the following functions. Note that the calls to **eq**, **neq**, and **getvar** could be removed and a single-level function could be used.

```
(eq (getvar <pattern> <field>) <value>)
(neq (getvar <pattern> <field>) <value>)
```

The pattern primitive

```
=(expression)
```

is replaced with
(eq (getvar <pattern> <field>) (expression))

in the join network and with

(eq (getfield <pattern>) (expression))

in the pattern network. For an inequality comparison (i.e., \(\sim\)), \texttt{neq} and \texttt{neqfield} can be used. The primitive expression

:(expression)

is replaced with

(expression)

The join network uses \texttt{getvar} calls to resolve references, and the pattern network uses \texttt{getfield} to resolve references. For a constraint used in conjunction with the \(\sim\) connective constraint (e.g., \(\sim:\)), the \texttt{not} function can be wrapped around the expression.

**GLOBAL VARIABLES**

<table>
<thead>
<tr>
<th>PTR_AND</th>
<th>PURPOSE: A pointer to the data structure for the \texttt{and} function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTR_CONSTANT</td>
<td>PURPOSE: A pointer to the data structure for the \texttt{constant} function.</td>
</tr>
<tr>
<td>PTR_EQ</td>
<td>PURPOSE: A pointer to the data structure for the \texttt{eq} function.</td>
</tr>
<tr>
<td>PTR_EQ_FIELD</td>
<td>PURPOSE: A pointer to the data structure for the \texttt{eq_field} function.</td>
</tr>
<tr>
<td>PTR_GET_FIELD</td>
<td>PURPOSE: A pointer to the data structure for the \texttt{get_field} function.</td>
</tr>
<tr>
<td>PTR_NEQ</td>
<td>PURPOSE: A pointer to the data structure for the \texttt{neq} function.</td>
</tr>
</tbody>
</table>
PTR_NEQ_FIELD

PURPOSE: A pointer to the data structure for the neq_field function.

PTR_NOP

PURPOSE: A pointer to the data structure for the nop function.

PTR_NOT

PURPOSE: A pointer to the data structure for the not function.

PTR_NOTCONSTANT

PURPOSE: A pointer to the data structure for the notconstant function.

PTR_OR

PURPOSE: A pointer to the data structure for the or function.

INTERNAL VARIABLES

None.

GLOBAL FUNCTIONS

CombineExpressions

PURPOSE: Combines two expressions into a single equivalent expression. Mainly serves to merge expressions containing and and or expressions without unnecessary duplication of the and and or expressions (i.e., two and expressions can be merged by placing them as arguments within another and expression, but it is more efficient to add the arguments of one of the and expressions to the list of arguments for the other and expression).

ARGUMENTS: Two expressions.

RETURNS: An expression.

OTHER NOTES: Modifies argument expressions to produce the final expression, so the original expressions are no longer valid after a call to this function. Null expressions are properly handled.
### GenAnd

**PURPOSE:** Generates an `and` function call with no arguments.

**RETURNS:** An expression.

### GenConstant

**PURPOSE:** Produces a constant (such as a symbol, integer, or function call stub).

**ARGUMENTS:** The type and value of the constant.

**RETURNS:** An expression.

### GenFourIntegers

**PURPOSE:** Generates an argument list consisting of four integers.

**ARGUMENTS:** Four integer indices.

**RETURNS:** An expression.

### GenGetfield

**PURPOSE:** Produces an expression of the form `(getfield <field-index>)`.

**ARGUMENTS:** Field index.

**RETURNS:** An expression.

### GenGetvar

**PURPOSE:** Produces an expression of the form `(getvar <pattern-index> <field-index>)`.

**ARGUMENTS:** Pattern and field indices.

**RETURNS:** An expression.

### GenGetvarValue

**PURPOSE:** Produces the integer indices for a getvar call.

**ARGUMENTS:** Pattern and field indices.

**RETURNS:** A void pointer value containing the encoded integer indices.
**GenJNColon**

**PURPOSE:** Generates a join network expression for testing a predicate constraint. The subfield `<function-call>` is converted to the expression `<function-call>` and the subfield `~:<function-call>` is converted to the expression (not `<function-call>`). References to variables in the expression are replaced with **getvar** calls.

**ARGUMENTS:** A flag indicating whether the subfield value is negated, the `<function-call>` associated with the subfield, the field and pattern indices of the subfield to be tested, and a flag indicating whether forward references to variables are allowed in the expression (for deftemplate patterns only).

**RETURNS:** An expression.

**GenJNConstant**

**PURPOSE:** Generates a join network expression for use in comparing subfield values to constants. The subfield `<value>` is converted to the expression (eq (getvar `<pattern-index>` `<field-index>`) `<value>`) and the subfield `~<value>` is converted to the expression (neq (getvar `<pattern-index>` `<field-index>`) `<value>`).

**ARGUMENTS:** A flag indicating whether the subfield value is negated, the type and value of the subfield, and the field and pattern indices of the subfield to be tested.

**RETURNS:** An expression.

**GenJNEq**

**PURPOSE:** Generates a join network expression for testing a return value constraint. The subfield `=<function-call>` is converted to the expression (eq (getvar `<pattern-index>` `<field-index>` `<function-call>`) and the subfield `~=<function-call>` is converted to the expression (neq (getvar `<pattern-index>` `<field-index>`) `<function-call>`). References to variables in the expression are replaced with **getvar** calls.

**ARGUMENTS:** A flag indicating whether the subfield value is negated, the `<function-call>` associated with the subfield, the field and pattern indices of the subfield to be tested, and a flag indicating whether forward references to variables are allowed in the expression (for deftemplate patterns only).

**RETURNS:** An expression.
**GenJNVariableComparison**

**PURPOSE:** Generates a join network expression testing the equality or inequality of variables bound to the fields of a pattern. Produces expressions of the form (eqvars <field-index-1> <pattern-index-2> <field-index-2>) when two fields must be equal and (neqvars <field-index-1> <pattern-index-1> <field-index-2>) when two fields must be unequal. The pattern associated with <field-index-1> in the comparison is assumed to be the pattern entering from the RHS of the join in which the expression is located. The depth field of the join structure is used to determine this pattern index.

**ARGUMENTS:** A flag indicating whether the variable is negated, the name of the variable, the pattern and field indices representing the pattern and field in which the variable was found, and a flag indicating whether forward references to variables are allowed in the expression (for deftemplate patterns only).

**RETURNS:** An expression.

**GenOr**

**PURPOSE:** Generates an or function call with no arguments.

**RETURNS:** An expression.

**GenPNColon**

**PURPOSE:** Generates a pattern network expression for testing a predicate constraint. The subfield :<function-call> is converted to the expression <function-call> and the subfield ~:<function-call> is converted to the expression (not <function-call>). References to variables in the expression are replaced with getfield calls.

**ARGUMENTS:** A flag indicating whether the subfield value is negated, the <function-call> associated with the subfield, and the field and pattern indices of the subfield to be tested.

**RETURNS:** An expression.

**GenPNConstant**

**PURPOSE:** Generates a pattern network expression for use in comparing subfield values to constants. The subfield <value> is converted to the expression (constant <value>) and the subfield ~<value> is converted to the expression (notconstant value).
ARGUMENTS: A flag indicating whether the subfield value is negated and the type and value of the subfield.

RETURNS: An expression.

---

**GenPNEq**

PURPOSE: Generates a pattern network expression for testing a return value constraint. The subfield =<function-call> is converted to the expression (eq (getfield <field-index>) <function-call>) and the subfield ~=<function-call> is converted to the expression (neq (getfield <field-index>) <function-call>). References to variables in the expression are replaced with getfield calls.

ARGUMENTS: A flag indicating whether the subfield value is negated, the <function-call> associated with the subfield, and the field and pattern indices of the subfield to be tested.

RETURNS: An expression.

---

**GenPNVariableComparison**

PURPOSE: Generates a pattern network expression testing the equality or inequality of variables bound to the fields of a pattern. Produces expressions of the form (eqfield <field-index-1> <field-index-2>) when two fields must be equal and (neqfield <field-index-1> <field-index-2>) when two fields must be unequal.

ARGUMENTS: A flag indicating whether the variable is negated, the name of the variable, and the pattern and field indices representing the pattern and field in which the variable was found.

RETURNS: An expression.

---

**GenTwoIntegers**

PURPOSE: Generates an argument list consisting of two integers.

ARGUMENTS: Two integer indices.

RETURNS: An expression.

---

**GetfieldReplace**

PURPOSE: Replaces variable references in an expression with appropriate getfield calls.
ARGUMENTS: A pointer to the expression to be modified and pattern and field indices representing the pattern and field from which the expression was extracted.

RETURNS: Nothing; however, the expression passed as a parameter is modified.

GetvarReplace

PURPOSE: Replaces variable references in an expression with appropriate getvar calls.

ARGUMENTS: A pointer to the expression to be modified, the pattern and field indices representing the pattern and field from which the expression was extracted, and a flag indicating whether forward references to variables are allowed in the expression (for deftemplate patterns only).

RETURNS: Nothing; however, the expression passed as a parameter is modified.

InitGenModule

PURPOSE: Initializes the global Function Pointers by calling FindFunction to locate each of the functions to be later referenced and setting the global value to the return value.

INTERNAL FUNCTIONS

None.
Build Module

Information and expressions generated during the analysis phase of rule compilation have to be integrated into the pattern and join networks. This integration takes advantage of the potential to share common expressions among patterns and joins where possible. The Build Module (build.c) uses information created by the Analysis Module and accessed through the Variable Manager Module to add a rule into the rule network consisting of the pattern and join networks.

The pattern network is conceptually represented as a tree structure. The root node represents the starting point of a pattern match before any elements of either the fact or pattern network have been “consumed.” The set of nodes after the root node represents all pattern expressions found as the first field in a pattern. The children of these nodes represent all second fields found in patterns. Each level of the pattern tree represents the set of all fields of a particular position in all patterns. As the pattern tree is traversed downward, fact fields are consumed as expressions are evaluated. Standard single field expressions consume one fact field when exited traversing downward in the tree. Multifield nodes consume all combinations of zero or more fact fields.

The pattern network structure allows patterns to share identical sequences of fields beginning at the front of the pattern. The two patterns

```
(data red ?)
(data green ?)
```

would share a common pattern node for their first field, data. If the pattern

```
(data green blue ?)
```

were now added, it would share the common pattern node, data, with the two patterns above. In addition, it would share the pattern node, green, with the second pattern.

For a given field in a pattern to be shared with a currently existing pattern field, two conditions must be met. First, all previous fields in the pattern must have been shared in the pattern network. Second, the expression generated for the field of the pattern must be identical to an expression already in place at the current level of addition in the pattern network. Note that variables generally do not create expressions that are tested in the pattern network unless they refer to a variable previously bound in the same pattern.

Sharing in the join network for a particular join of a rule occurs under three conditions. First, all previous joins must be shared. Second, the expression generated for the join must be identical to an expression already in place at the current level of addition in the join network. Third, the join to be shared must be entered from the same location in the pattern network. The following two rules illustrate some examples of sharing:

```
(defrule example1
 (data red ?x)
 (data green ?x)
 =>)
```

```
(defrule example2
 (data red ?x)
```

CLIPS Architecture Manual 143
Many examples of pattern sharing occur. All four patterns share a common node testing for the constant value data. In addition, the first pattern in both rules can share all pattern nodes. The join for the first pattern in both rules also can be shared. The second join for both rules, however, cannot be shared. The expression for the second join in both rules is identical (i.e., \((\text{eqvars} 2 \ 1 \ 2)\)); however, the joins must be entered from different patterns and, therefore, cannot be shared.

The following rule:

```
(deerule example3
  (data red ?y)
  (data blue ?y)
  (info ?z)
 =>)
```

would be able to share nodes in the pattern and join networks. Its first two patterns already exist in the pattern network. The third pattern would require the addition of two new nodes in the pattern network. The first two joins for this rule could be shared with the joins used for rule example2. A third join for rule example3 would have to be added for the last pattern. Note that the use of different variable names does not affect the ability to share. As long as the expressions generated are identical, sharing will occur. Variable names serve only as positional references.

Information about sharing in join network is displayed when a rule is being loaded if the watch compilations flag is on. New additions to the join network is signaled with +j and reuse of existing nodes is indicated with =j.

The CLIPS join topology differs slightly from the “standard” Rete topology used by OPS5. First, each pattern corresponds to its own join. In standard Rete topology, the first two patterns will form a two-input join. If only one pattern exists, this pattern will form a single one-input join. Thus, using this topology, the number of joins needed for n patterns is n - 1 with a minimum of 1 join. In CLIPS, the first pattern always creates a one-input join. This simplifies the algorithms used considerably since a pattern always enters the join from the RHS. Given this simplification, the beta memory of a join can never be associated with a pattern contained in not CE.

Standard Rete topology also makes a test in the pattern network for the length of a fact. The use of multifield variables in CLIPS eliminates much of the usefulness of making this test. Each level of the pattern network corresponds directly to specific indexed fields in a pattern.

**GLOBAL VARIABLES**

None.

**INTERNAL VARIABLES**

<table>
<thead>
<tr>
<th>PatternNetworkPointer</th>
</tr>
</thead>
</table>

**PURPOSE:** A pointer to the root node of pattern network. This provides access to the entire rule network.
GLOBAL FUNCTIONS

ConstructJoins

PURPOSE: Integrates a set of pattern and join tests associated with a rule into the pattern and join networks.

ARGUMENTS: A pointer to the pointer of the top node in the pattern network and a pointer to the structure storing information about the rule to be added.

RETURNS: A pointer to the last join that was added.

OTHER NOTES: Prints informational messages to indicate join network sharing. If a new node has to be added to the join network, +j is printed. If a join node can be reused, =j is printed. Information about pattern and join expressions is retrieved from the Variable Manager Module.

DetachJoins

PURPOSE: Removes a join node and all of its parent nodes from the join network. Nodes are only removed if they are no longer shared (i.e. a join that has more than one child node is shared). Any partial matches associated with the join are also removed. A rule's joins are typically removed by removing the bottom most join used by the rule and then removing any parent joins which are not shared by other rules.

ARGUMENTS: A pointer to a join node.

NetworkPointer

PURPOSE: Returns the value of the PatternNetworkPointer.

RETURNS: A pointer to a pattern node.

SetNetworkPointer

PURPOSE: Sets the value of the PatternNetworkPointer.

ARGUMENTS: A pointer to a pattern node.
INTERNAL FUNCTIONS

**DetachPattern**

**PURPOSE:** Removes a pattern node and all of its parent nodes from the pattern network. Nodes are only removed if they are no longer shared (i.e., a pattern node that has more than one child node is shared). A pattern from a rule is typically removed by removing the bottom most pattern node used by the pattern and then removing any parent nodes which are not shared by other patterns.

**ARGUMENTS:** A pointer to a pattern node.

**PlacePattern**

**PURPOSE:** Integrates a pattern into the pattern network.

**ARGUMENTS:** Current level in the pattern network at which the new pattern is being integrated, the previous level in the pattern network at which the new pattern has already been integrated, an integer index indicating which pattern from the rule is being integrated, an integer index indicating which field of the pattern is being integrated, an integer value indicating the number of fields in the pattern, and a pointer to the variable which points to the root node in the pattern network. Information about pattern expressions is obtained from the Variable Manager Module.

**RETURNS:** A pointer to the last pattern node in the pattern network for the pattern just added.

**RemoveIntranetworkLink**

**PURPOSE:** Removes the link between a join node in the join network and its corresponding pattern node in the pattern network. If the pattern node is then no longer associated with any other joins, it is removed using the function **DetachPattern**.

**ARGUMENTS:** A pointer to a join node.

**ReuseJoin**

**PURPOSE:** Determines whether a join exists that can be reused for the join currently being added to the join network.

**ARGUMENTS:** A pointer to the list of possible joins that can be reused, a flag indicating whether the join to be added is the first join for the rule, a flag indicating whether the pattern associated with
the join is contained with a not CE, the primary and secondary join expressions for the join to be added, and a pointer to the list of joins connected to the pattern associated with the join being added.

RETURNS: A pointer to the join that can be reused if one exists; otherwise NULL.
The Drive Module (drive.c) contains the major functionality for updating the join network when a fact has been asserted into the knowledge base. This update will also be referred to as “driving” a set of partial matches through the join network. When a fact has matched a pattern in the pattern network, a partial match consisting of that single fact is created. This partial match is then sent to each join in the join network connected to that pattern. The partial match “enters” through the RHS of the join. Depending upon the type of the join (i.e. associated with the first conditional element, a not conditional element, etc.), the entering partial match will be compared with the beta memory of the join. New partial matches may be created from this comparison and sent to the descendent joins of the current join. New partial matches would enter from the LHS of the descendent joins. Note that all pattern conditional elements enter joins from the RHS, but all joins enter other joins from the LHS. The algorithm for driving partial matches through the join network is described as follows.

The function Drive handles high-level updating of the join network when a fact is asserted. If the join being updated is a terminator join (i.e., the last join of a rule which connects the join to the actions of a rule), a rule activation has occurred. An activation is added to the agenda and the current level of recursive descent into the join network is terminated. If the join was entered from the LHS, the partial match is stored in the beta memory of the join. Partial matches entering from the RHS are already stored in the alpha memory in the pattern network.

If the join being updated is a single-entry join (i.e. the join associated with the first conditional element in a rule), then the single-entry join algorithm is used for updating the join network. First, it is determined if the primary join expression evaluates to TRUE or FALSE. If no expression exists, the evaluation is automatically TRUE. If the expression is FALSE, the join update is completed.

If LHS entry is from a pattern conditional element, a copy of the alpha partial match is made and sent to all the child joins of the current join using the Drive algorithm.

If LHS entry is from a not conditional element and the count of facts matching the CE is greater than zero, then the count associated with the join is incremented by one. In effect, other facts are already preventing the not CE from being satisfied, so just keep track of the fact that there is one more fact prevent the CE from being satisfied. If, however, the count was less than zero (indicating that previous facts had been asserted which matched the CE), the join’s count is set to one and all partial matches containing the pseudo-fact ID which was stored as the join’s count are removed from all of the descendent joins of the current join.

For double-entry joins, a loop is used to compare each set of partial matches in the opposite memory (beta memory for RHS entry and alpha memory for LHS entry) to the entering partial match. If the join was entered from the RHS, each partial match in the beta memory is compared to the entering partial match. If the join was entered from the LHS, each partial match in the alpha memory (stored in the pattern network) is compared to the entering partial match. For each pair of partial matches that is compared, the primary join expression is evaluated for its boolean value. If no expression exists, the evaluation is considered TRUE. If the join expression evaluates to TRUE, one of three algorithms is performed. The three algorithms correspond to the following cases: positive RHS entry and positive LHS entry, positive RHS entry and negative LHS entry (meaning the conditional element associated with this join is a not
conditional element) with the partial match entering from the LHS, and positive RHS entry and negative LHS entry with the partial match entering from the RHS. Algorithms for each of these cases are described below.

The double-entry join algorithm for joins with positive RHS entry and positive LHS entry works as follows. The beta partial match and alpha partial match are merged to form a new partial match with the alpha partial match attached to the end of the beta partial match. This new partial match is then sent to all the child joins of the current join using the Drive algorithm.

The double-entry join algorithm for joins associated with a not conditional element in which a partial match enters from the LHS works as follows. The count value associated with the beta partial match is incremented by one. This count is originally set to zero when the beta partial match enters the join. If it is still zero at completion of all memory comparisons, a partial match will be created by merging the beta partial match with a pseudo-fact ID.

The double-entry join algorithm for joins associated with a not conditional element in which a partial match enters from the RHS works as follows. If the count associated with the beta memory partial match is greater than zero, increment the count by one. If the count was less than zero, set the beta memory count to one and remove all partial matches containing the pseudo-fact ID previously associated with the beta memory partial match from all of the descendent joins of the current join.

Upon completion of the alpha and beta memory comparisons, a final test is performed. If the RHS join entry is associated with a not conditional element, the join was entered from the LHS, and the number of alpha memory partial matches which satisfied the primary join expression was zero, then a new partial match may need to be created. The secondary join expression is evaluated. If it evaluates to TRUE, a partial match consisting of a pseudo-fact identification (ID) number and the beta memory partial match is created. The pseudo-fact identification ID number is negative. The count value of the beta memory partial match is set to this pseudo-fact ID. The count value represents the number of alpha memory partial matches which satisfied the primary join expression for a particular beta memory partial match. A negative value indicates that no alpha partial matches satisfied the join expression. The new partial match create from the pseudo-fact ID and the beta memory partial match is sent to all the descendent joins of the current join using the Drive algorithm. Note that, if the beta memory count had been greater than zero, this would have indicated that facts in the alpha memory of the join conflicted with the beta memory and prevented a partial match for this join.

As an example, consider the following rule:

\[(\text{defrule match "}"
\text{(point ?x ?)}
\text{(point ? ?x)}
\text{=>})\]

The following diagram illustrates the join network for this rule. The two boxes represent joins. The top box is the join associated with the first pattern CE (point ?x ?). The bottom box is the join associated with the second pattern CE (point ? ?x). The line terminated with a dark circle to the left of each join represents the contents of the beta memory of the joins. As shown here, the beta memory for both joins is empty. The circle represents the pattern node signifying the completion of the pattern (point ? ?). This pattern is used for both (point ?x ?) and (point ? ?x) since variables in this case cannot
be checked in the pattern network. The alpha memory is represented by the line terminated with a dark circle connected to the right of the pattern node. The alpha memory is also empty. Note that the expressions associated with each join are not shown. The top join has no expression and will allow any partial match to filter down to the next join. The bottom join must verify that the value of the second field of the fact bound to the first pattern is equal to the value of the third field of the fact bound to the second pattern. This diagram assumes that the match rule is the only rule in the rule network (hence, no sharing). Count values for partial matches are not shown since these are applicable only to joins that are attached to not CEs. The terminator join for the rule is not shown in the diagram. This last join of the rule stores the partial matches which satisfy all CEs of the rule. A link between these partial matches and their corresponding activations on the agenda is also maintained.

Suppose that the following fact were entered into the fact-list:

\[ f-1 \quad (\text{point } 3 \ 4) \]

This fact would match the pattern \((\text{point } ? \ ?)\) and be placed in the alpha memory of the pattern node. The pattern node must then pass this partial match to the two joins to which it is connected. First, the partial match is sent to the top join as shown below.

\[ (\text{point } ? \ ?) \]

\[ f-1 \]
Since the top join has no expression, the partial match is sent down to the next join.

\[
\text{(point ? ?)}
\]

A comparison now takes place between the partial match in the beta memory and all partial matches in the alpha memory associated with this join. Currently, only one partial match is in the alpha memory. For the second join, the partial match in the alpha memory is the fact bound to the first pattern, and the partial match in the beta memory is the fact bound to the second pattern. Evaluating the join expression will produce a FALSE result since the value bound to \( ?x \) in the first pattern is 3 and the value bound to \( ?x \) in the second pattern is 4.

\[
\text{(point ? ?)}
\]

The process of updating the top join is complete. Now the partial match in the alpha memory is sent to the bottom join.
Once again, a comparison is made between the facts to check proper variable bindings. As before, this comparison will fail.

Suppose that an additional fact were entered into the fact-list.

\[ f-2 \quad (\text{point 4 3}) \]

This fact would match the pattern \((\text{point ? ?})\) and be placed in the alpha memory of the pattern node. The pattern node must then pass this partial match to the two joins to which it is connected. First, the partial match is sent to the top join as shown following.
Since the top join has no expression, the partial match is sent down to the next join.

A comparison now takes place between the partial match in the beta memory and all partial matches in the alpha memory associated with this join. The first comparison is between f-2 in beta memory and f-1 in alpha memory. This comparison succeeds. The value of ?x in f-1 (the second field) is 3 and the value of ?x in f-2 (the third field) is also 3. A partial match consisting of f-1 and f-2 is created and sent to the next join. Since this next join is a terminator join, the rule that is matched has been satisfied with f-1 bound to the first pattern and f-2 bound to the second pattern. This activation would be placed on the agenda.
The next comparison now takes place. The value of ?x in the first pattern (the second field of f-2) is 4; however, the value of ?x in the second pattern (the third field of f-2) is 3. The comparison fails and no new partial match is created.

The process of updating the top join is complete. Now the partial match in the alpha memory is sent to the bottom join.
A comparison now takes place between the partial match in the alpha memory associated with this join and all partial matches in the beta memory of the join. The first comparison is between f-1 in beta memory and f-2 in alpha memory. This comparison succeeds. The value of ?x in the first pattern (the second field of f-2) is 4 and the value of ?x in the second pattern (the third field of f-1) is also 4. A partial match consisting of f-2 and f-1 is created and sent to the next join. Since this next join is a terminator join, the rule that is matched has been satisfied with f-2 bound to the first pattern and f-1 bound to the second pattern. This activation would be placed on the agenda.

The next comparison now takes place. The value of ?x in the first pattern (the second field of f-2) is 4; however, the value of ?x in the second pattern (the third field of f-2) is 3. The comparison fails and no new partial match is created.
The process of updating the bottom join for the addition of f-2 is complete. Notice that the beta memory of the top join never contains any partial matches. Joins that correspond to the first pattern of a rule will never make use of the beta memory.

GLOBAL VARIABLES

<table>
<thead>
<tr>
<th>IncrementalReset</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IncrementalResetFlag</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>

INTERNAL VARIABLES

None.

GLOBAL FUNCTIONS

<table>
<thead>
<tr>
<th>Drive</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
<tr>
<td>ARGUMENTS:</td>
</tr>
<tr>
<td>OTHER NOTES:</td>
</tr>
</tbody>
</table>
**PNRDrive** as necessary to process the addition of the partial match.

---

### EvaluateJoinExpression

**PURPOSE:**
Evaluates join expressions. Performs a faster evaluation for join expressions than if **EvaluateExpression** were used directly. Function calls to **eq_vars**, **neq_vars**, **and**, and **or** are evaluated directly. All other function calls are evaluated using **EvaluateExpression**.

**ARGUMENTS:**
A pointer to the expression to be evaluated, pointers to the partial matches from the alpha and beta memory associated with the expression, and a pointer to the join associated with the expression.

**RETURNS:**
Boolean value. The result of the evaluation of the expression.

---

### GetIncrementalReset

**PURPOSE:**
Returns the current value of the **IncrementalReset** flag.

**RETURNS:**
A boolean value.

---

### PNLDirect

**PURPOSE:**
Handles the entry of a partial match from the LHS of a join that has positive LHS entry and negative RHS entry (meaning the conditional element associated with this join is a **not** conditional element). An new partial match is created by combining the match from the beta memory with a "pseudo" partial match corresponding to the facts which didn't match the **not** CE. Once merged, the new partial match is sent to each child join of the join from which the merge took place.

**ARGUMENTS:**
A pointer to the join being processed and a pointer to the partial match from the join's beta memory that entered from the LHS of the join.

---

### SetIncrementalReset

**PURPOSE:**
Sets the current value of the **IncrementalReset** flag.

**ARGUMENTS:**
A boolean value (the new value of the flag).

**RETURNS:**
A boolean value (the old value of the flag).
INTERNAL FUNCTIONS

ClearLowerBetaMemory

PURPOSE: Removes all partial matches from the beta memory of a join and all joins which descend from that join.

ARGUMENTS: A pointer to the join at which the removal of partial matches should begin.

EmptyDrive

PURPOSE: Handles the entry of a alpha memory partial match from the RHS of a join that is the first join of a rule (i.e. a join that cannot be entered from the LHS). Both positive and negative RHS join entry are handled.

ARGUMENTS: A pointer to the join being processed and a pointer to the partial match from the join’s alpha memory that entered from the RHS of the join.

JoinNetErrorMessage

PURPOSE: Prints an informational message indicating which join of a rule generated an error when a join expression was being evaluated.

ARGUMENTS: A pointer to the join being processed when the error occurred.

PNRDrive

PURPOSE: Handles the entry of a partial match from the RHS of a join that has positive LHS entry and negative RHS entry (meaning the conditional element associated with this join is a not conditional element). Entry of the alpha memory partial match will cause the count value of the associated beta memory partial match to be incremented. This in turn may cause partial matches associated with the beta memory partial match to be removed from the network.

ARGUMENTS: A pointer to the join being processed and a pointer to the partial match from the join's beta memory that entered from the LHS of the join.

PPDrive

PURPOSE: Handles the merging of an alpha memory partial match with a beta memory partial match for a join that has positive LHS
entry and positive RHS entry. The partial matches being merged have previously been checked to determine that they satisfy the constraints for the join. Once merged, the new partial match is sent to each child join of the join from which the merge took place.

ARGUMENTS: Pointers to the partial matches from the alpha and beta memory being merged and a pointer to the join from which the partial matches originated.
The Engine Module (engine.c) provides functionality for browsing, maintaining, updating, and executing the agenda.

GLOBAL VARIABLES

**AgendaChanged**

PURPOSE: Boolean flag. If TRUE, indicates that the Agenda has been altered. Updated to TRUE whenever an activation is added to, removed from, or moved on the Agenda.

**DeletedFiringRule**

PURPOSE: Boolean value. If TRUE, the currently executing rule has been deleted.

**ExecutingRule**

PURPOSE: A pointer to the rule information data structure of the currently executing rule. If NULL, then no rules are executing.

**HaltRules**

PURPOSE: Boolean value. If TRUE, rule execution should be halted.

**TheLogicalJoin**

PURPOSE: A pointer to the join for a rule at which the partial matches needed to set up logical dependencies are stored. If a rule contains no logical conditional elements, then this value is NULL.

INTERNAL VARIABLES

**Agenda**

PURPOSE: Pointer to the list of rule activations which have not yet fired.

**CurrentTimetag**

PURPOSE: Integer value used to provide a unique identification number for each activation added to the Agenda. Initially zero, this value is incremented by one each time an activation is added to the agenda.
### ListOfRunFunctions

**PURPOSE:** A list of functions which are to be executed after each rule firing.

### NumberOfActivations

**PURPOSE:** Integer value representing the number of activations currently on the Agenda.

### RuleFiring

**PURPOSE:** A pointer to the name of the currently executing rule.

### SalienceEvaluation

**PURPOSE:** An integer value representing the current type of salience evaluation (either when defined, when activated, or every cycle).

### Strategy

**PURPOSE:** An integer value representing the current conflict resolution strategy (either depth, breadth, lex, mea, complexity, simplicity, or random).

### WatchActivations

**PURPOSE:** Boolean flag. When TRUE, enables printing of messages indicating addition and removal of activations to the Agenda.

### WatchStatistics

**PURPOSE:** Boolean flag. When TRUE, statistical information such as the number of rule firings is printed after the run command is executed.

### GLOBAL FUNCTIONS

#### ActivationBasis

**PURPOSE:** Returns the partial match associated with an activation.

**ARGUMENTS:** A generic pointer to an activation.

**RETURNS:** A pointer to a partial match.
**AddActivation**

PURPOSE: Creates a rule activation to be added to the Agenda and links the activation with its associated partial match. The function PlaceActivation is then called to place the activation on the Agenda. Typically called when all patterns on the LHS of a rule have been satisfied.

ARGUMENTS: The last join of the rule associated with the activation and a pointer to the partial match which activated the rule.

**AddBreakpoint**

PURPOSE: Adds a breakpoint for the specified rule.

ARGUMENTS: A generic pointer to a defrule structure.

**AddRunFunction**

PURPOSE: Adds a function to the ListOfRunFunctions.

ARGUMENTS: A name to be associated with the function, a pointer to the function, and the priority of the run item.

**ClearRuleFromAgenda**

PURPOSE: Removes all activations of a specified rule from the Agenda.

ARGUMENTS: Name of the rule.

**DefruleHasBreakpoint**

PURPOSE: Indicates whether the specified rule has a breakpoint set.

ARGUMENTS: A generic pointer to a defrule structure.

RETURNS: Boolean value. TRUE if the defrule has a breakpoint set, otherwise FALSE.

**DeleteActivation**

PURPOSE: Deletes the specified activation from the agenda.

ARGUMENTS: A pointer to an activation or NULL to remove all activations.

RETURNS: Boolean Value. TRUE, if the specified activation exists and was deleted from the agenda, otherwise FALSE.
GetActivationName

PURPOSE: Returns the name of the rule associated with an activation.

ARGUMENTS: A generic pointer to an activation.

RETURNS: The name of a rule.

GetActivationPPForm

PURPOSE: Returns the pretty print representation of an activation.

ARGUMENTS: A character buffer in which to store the pretty print representation, the size of the buffer in characters, and a generic pointer to an activation.

RETURNS: No return value. The pretty print representation is stored in the character buffer passed as an argument.

GetActivationSalience

PURPOSE: Returns the salience value of an activation.

ARGUMENTS: A generic pointer to an activation.

RETURNS: An integer value.

GetAgendaChanged

PURPOSE: Returns the value of the variable AgendaChanged.

RETURNS: Boolean value (TRUE or FALSE).

GetNextActivation

PURPOSE: Returns an activation from the Agenda.

ARGUMENTS: A generic pointer to an activation. If the pointer is NULL, the first activation in the Agenda is returned. If the pointer is not NULL, the next activation after the pointer is returned.

RETURNS: A generic pointer to an activation. A NULL pointer indicates that there are no further activations in the Agenda.

GetNumberOfActivations

PURPOSE: Returns the value of the variable NumberOfActivations.

RETURNS: An integer value.
GetRuleFiring

PURPOSE: Returns the value of the variable RuleFiring.
RETURNS: The name of a rule.

GetSalienceEvaluation

PURPOSE: Returns the value of the variable SalienceEvaluation.
RETURNS: An integer value.

GetStrategy

PURPOSE: Returns the value of the variable Strategy.
RETURNS: An integer value.

InitializeEngine

PURPOSE: Initializes the activations and statistics watch items.

ListAgenda

PURPOSE: Lists all of the activations on the agenda to the logical name wdisplay.

ListBreakpoints

PURPOSE: Lists all of the breakpoints to the logical name wdisplay.

MoveActivationToTop

PURPOSE: Moves the specified activation in the agenda to the top of the agenda.
ARGUMENTS: A pointer to an activation.
RETURNS: Boolean Value. TRUE, if the specified activation exists and was moved to the top of the agenda, otherwise FALSE.

PrintActivation

PURPOSE: Prints an activation in a “pretty” format. Salience, rule name, and the partial match which activated the rule are printed.
ARGUMENTS: Logical name to which output is sent and a pointer to an activation.
PrintCRSVActivation

PURPOSE: Prints an activation in a CRSV trace file compatible format.

ARGUMENTS: Logical name to which output is sent and a pointer to an activation.

RefreshAgenda

PURPOSE: Recomputes the salience values for all activations on the Agenda and then reorders the Agenda.

RemoveActivation

PURPOSE: Returns an activation and its associated data structures to the Memory Manager. Links to other activations and partial matches may also be updated.

ARGUMENTS: A pointer to an activation, a flag indicating whether the links between activations on the agenda should be updated, and a flag indicating whether the links between the activation and its corresponding partial match should be updated.

ReorderAgenda

PURPOSE: Reorders the Agenda based on the current conflict resolution strategy.

RemoveAllActivations

PURPOSE: Removes all activations from the Agenda.

RemoveAllBreakpoints

PURPOSE: Removes all breakpoints.

RemoveBreakpoint

PURPOSE: Removes a breakpoint for the specified rule.

ARGUMENTS: A generic pointer to a defrule structure.

RETURNS: Boolean value. TRUE if the breakpoint was found and removed, otherwise FALSE.

RemoveRunFunction

PURPOSE: Removes a function from the ListOfRunFunctions.
ARGUMENTS: Name associated with the run function.

**RunCLIPS**

PURPOSE: Begins execution of rules on the Agenda.

ARGUMENTS: An integer representing the maximum number of rules that can be fired. If run limit is less than zero, rules will be executed until the agenda is empty. If run limit is greater than zero, rules will be executed until either the Agenda is empty or the run limit has been reached.

RETURNS: Number of rules fired.

**SetActivationSalience**

PURPOSE: Sets the salience value of an activation.

ARGUMENTS: A generic pointer to an activation and the new salience value.

RETURNS: The old salience value.

**SetAgendaChanged**

PURPOSE: Sets the value of the variable AgendaChanged.

ARGUMENTS: Boolean value (TRUE or FALSE).

**SetSalienceEvaluation**

PURPOSE: Sets the value of the variable SalienceEvaluation.

ARGUMENTS: An integer value representing the new type of salience evaluation (either when defined, when activated, or every cycle).

RETURNS: An integer value representing the old type of salience evaluation.

**SetStrategy**

PURPOSE: Sets the value of the variable Strategy and then calls the ReorderAgenda function to update the Agenda.

ARGUMENTS: An integer value representing the new conflict resolution strategy (either depth, breadth, lex, mea, complexity, simplicity, or random).
INTERNAL FUNCTIONS

**CompareBindings**

PURPOSE: Compares two activations using the lex conflict resolution strategy to determine which activation should be placed first on the agenda. This lexicographic comparison function is used for both the lex and mea strategies.

ARGUMENTS: Two pointers to the activations to be compared.

RETURNS: An integer value indicating whether the first activation has higher, lesser priority, or equivalent priority to the second activation.

**PlaceActivation**

PURPOSE: Coordinates placement of an activation on the Agenda based on the current conflict resolution strategy.

ARGUMENTS: A pointer to an activation.

**PlaceBreadthActivation**

PURPOSE: Determines where an activation should be placed on the Agenda for the breadth conflict resolution strategy.

ARGUMENTS: A pointer to an activation.

RETURNS: A pointer to an activation already on the Agenda after which the new activation should be placed. If NULL, then the activation should be placed at the beginning of the Agenda.

**PlaceComplexityActivation**

PURPOSE: Determines where an activation should be placed on the Agenda for the complexity conflict resolution strategy.

ARGUMENTS: A pointer to an activation.

RETURNS: A pointer to an activation already on the Agenda after which the new activation should be placed. If NULL, then the activation should be placed at the beginning of the Agenda.
### PlaceDepthActivation

**PURPOSE:** Determines where an activation should be placed on the **Agenda** for the depth conflict resolution strategy.

**ARGUMENTS:** A pointer to an activation.

**RETURNS:** A pointer to an activation already on the **Agenda** after which the new activation should be placed. If NULL, then the activation should be placed at the beginning of the **Agenda**.

### PlaceLEXActivation

**PURPOSE:** Determines where an activation should be placed on the **Agenda** for the lex conflict resolution strategy.

**ARGUMENTS:** A pointer to an activation.

**RETURNS:** A pointer to an activation already on the **Agenda** after which the new activation should be placed. If NULL, then the activation should be placed at the beginning of the **Agenda**.

### PlaceMEAAActivation

**PURPOSE:** Determines where an activation should be placed on the **Agenda** for the mea conflict resolution strategy.

**ARGUMENTS:** A pointer to an activation.

**RETURNS:** A pointer to an activation already on the **Agenda** after which the new activation should be placed. If NULL, then the activation should be placed at the beginning of the **Agenda**.

### PlaceRandomActivation

**PURPOSE:** Determines where an activation should be placed on the **Agenda** for the random conflict resolution strategy.

**ARGUMENTS:** A pointer to an activation.

**RETURNS:** A pointer to an activation already on the **Agenda** after which the new activation should be placed. If NULL, then the activation should be placed at the beginning of the **Agenda**.

### PlaceSimplicityActivation

**PURPOSE:** Determines where an activation should be placed on the **Agenda** for the simplicity conflict resolution strategy.
ARGUMENTS: A pointer to an activation.

RETURNS: A pointer to an activation already on the Agenda after which the new activation should be placed. If NULL, then the activation should be placed at the beginning of the Agenda.

SortBindings

PURPOSE: Copies a partial match and then sorts the fact-indices in the copied partial match in ascending order.

ARGUMENTS: A pointer to a partial match.

RETURNS: A copied version of the partial match with sorted fact-indices.
The Match Module (match.c) contains the functionality necessary for traversing the pattern network. The pattern network determines which patterns a fact has matched. The pattern network is organized as a tree structure. Pattern network levels correspond directly to the sequential order of fields found in patterns. That is, all pattern constraints occurring in the first field of a pattern are found in the first level of the pattern network. All pattern constraints that occur in the second field of a pattern are found in the second level of the pattern network. The leaf nodes of the pattern network represent the end of a pattern. These leaf nodes connect the pattern network to the join network.

Each pattern node contains several pieces of information. A pointer is stored to the next and previous sibling nodes as well as a pointer to the parent node and the first child node. The child nodes of a pattern node can be determined by following the first child node value and then following the next sibling node value of each child. Each pattern node contains information about whether it is intended to match a single field or multiple fields of a fact. Only multifield variables and wildcards can match multiple fields. Each node may also contain an expression which, when evaluated, determines whether a field has satisfied a pattern constraint. In addition, an end-of-pattern leaf node contains a pointer to an alpha memory which stores a list of all facts that matched the pattern and a pointer to the list of joins in the join network that are to be entered from the leaf node when a fact has matched the pattern.

As an example, the following patterns (possibly found in one or more rules)

(fact $?)
(fact $?x red $?y)
(item ?x)
(item ?y)

would produce the following pattern network:
Notice that the patterns (item ?x) and (item ?y) are treated as the same pattern in the pattern network. The patterns (fact $?x) and (fact $?x red $?y) share their first two fields in the pattern network. The left-most “end of pattern” node is the node associated with the successful pattern match of the pattern (fact $?x). The middle “end of pattern” node is the node associated with the successful pattern match of the pattern (fact $?x red $?y). The right-most “end of pattern” node is the node associated with the successful pattern match of the patterns (item ?x) and (item ?y).

When a new fact is added to the fact-list, it must traverse the pattern network to determine which patterns it has matched. A traversal of the pattern network must be complete. That is, all patterns that have been matched must be found. Traversal involves testing fields of the fact against pattern expressions found in the pattern network. As stated previously, the first level of the pattern network performs tests for the first fields of the patterns, the second level of the pattern network performs tests for the second fields of the patterns, and so on. In general, this means that the first level of the pattern network performs tests against the first field of the newly asserted fact, the second level of the pattern network performs tests against the second field of the newly asserted fact, and so on. Pattern nodes that can match against multiple fields can throw this strict correspondence off. Levels below a multifield pattern node do not correspond directly to a field in the fact; however, they still correspond directly to a field in the pattern.

Pattern network traversal begins by assigning one pointer the value of the first field of the newly asserted fact. Hereafter, this pointer will be called the fact field pointer. Another pointer is assigned the value of the “upper-left,” “top-most,” or “root” node in the pattern network. Hereafter, this pointer will be called the current pattern pointer.

If the pattern node is intended to match a single field and no expression is associated with the pattern node, the fact field pointer is “incremented” to point to the next field of the fact. Matching then proceeds to the first child node of the current pattern node.

If the single-field pattern node has an expression associated with it, that expression must be evaluated. If the expression evaluates to true, the fact field pointer is “incremented” and matching proceeds to the first child node of the current pattern node. If the expression evaluates to false, the matching attempt for this pattern node has failed and backtracking must take place. For example, if the expression associated with a pattern node was (constant red), the matching process could only proceed past this pattern node if the field pointed to by the fact field pointer had the value of red.

If at any point either the current pattern node is a leaf node and the fact field pointer is pointing at a remaining field in the fact or the current pattern node is not a leaf node and the fact field pointer is empty (i.e., no more fields are left in the fact), the matching process has failed. The length of the fact does not match the length of the pattern currently being matched. Backtracking takes place when this occurs.

If a leaf node is reached and the fact field pointer is empty, a pattern has been matched. The fact id of the fact being matched is stored in the alpha memory of the pattern node and the partial match containing the single-fact id is sent to all joins in the join network connected to this pattern node. Backtracking then takes place to find other pattern matches.

Multifield variables and wildcards add another level of complexity to the pattern matching process. The process for matching a multifield pattern node is similar to the single-field pattern nodes, with the exception that there is usually more than one way in which a multifield pattern node can match against a fact. To accommodate multifield
matches (which match multifield pattern nodes to zero or more fields of a fact), the pattern matching algorithm is entered recursively for each of all possible ways in which the multifield node can match. For example, if a multifield node is entered and two fields remain in the fact, the match algorithm is recursively entered three times with the multifield being bound respectively to zero, one, and two fields of the fact.

Multifield markers are used to keep track of the fields matched by a multifield variable or wildcard. That is, if a multifield variable were the third field of a pattern, a multifield marker might contain the information “The third field of the pattern matched the third through sixth fields of the fact.” Multifield markers are chained together as multifield pattern nodes are encountered.

If a failure ever occurs while pattern matching or a leaf node has been successfully reached, backtracking must occur in an attempt to find other matches. If the current pattern node has a right-sibling node, the current pattern node is set to the right-sibling node. Otherwise, continue setting the current pattern node to the parent of the current pattern node until a node is reached that has a right sibling. For each level backtracked, the fact-field pointer and pattern field pointer need to be decremented. If there is no pattern node (i.e., the parent of the root node) or a multifield node is reached, a return should be made from this recursive level of the pattern network traversal. Multifield markers are unchained when returning from a level of recursion in the pattern matching process.

The pattern matching process also makes use of some shortcuts to increase speed. For example, all expression that test for field equality against a single constant value are placed toward the “right” or “end” of the list of siblings. Whenever a test against one of these nodes succeeds, it is not necessary to backtrack to the right-sibling nodes since the tests for these nodes are mutually exclusive. In addition, a multifield node that has only a single “end of pattern” node as a child does not have to generate all possible matches. The multifield node must bind to all remaining fields in the fact.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

<table>
<thead>
<tr>
<th>CurrentPatternFact</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: A pointer to the fact currently being matched.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CurrentPatternMarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: A list of multifield markers for the fact currently being matched.</td>
</tr>
</tbody>
</table>
GLOBAL FUNCTIONS

GetFieldSysFunction

PURPOSE: Extracts a specified field value from a fact during the pattern matching process for the purpose of expression evaluation. This is the C implementation of the `getfield` function discussed in the Generate Module.

ARGUMENTS: A pointer to a DATA_OBJECT structure in which the field value will be stored.

RETURNS: Value of the variable extracted from the fact

OTHER NOTES: Uses the global variables `CurrentPatternFact` and `CurrentPatternMarks` to extract the value.

PatternMatch

PURPOSE: Filters a fact through the pattern network searching for all patterns matches by the fact.

ARGUMENTS: A pointer to the fact being matched, a pointer to the current element in the fact being matched, a pointer to the pattern node being matched against, an index for the depth traversed into the pattern network, an index to the current field being matched in the fact, a pointer to the list of multifield markers, and a pointer to the last multifield marker in the list of multifield markers.

OTHER NOTES: Some parameters provide redundant information. See above for description of compare algorithm.

PatternNetErrorMessage

PURPOSE: Prints an error message when a error occurs as the result of evaluating an expression in the pattern network. Prints the fact currently being matched against and the field in the pattern which caused the problem, then calls the function `TraceErrorToPattern` to further isolate the error.

ARGUMENTS: A pointer to the pattern node being matched against when the error occurred.

TraceErrorToRule

PURPOSE: Prints an error message when a error occurs as the result of evaluating an expression in the pattern network. Used to indicate which rule caused the problem.
ARGUMENTS: A pointer to the join node associated with the pattern being matched against when the error occurred and an integer value indicating the number of spaces that should be printed before the rule name.

LOCAL FUNCTIONS

### CopySegmentMarkers

**PURPOSE:** Copies a list of multifield markers.

**ARGUMENTS:** A pointer to a list of multifield markers.

**RETURNS:** A pointer to a copied list of multifield markers.

### EvaluatePatternExpression

**PURPOSE:** Evaluates an expression found in a node the pattern network which is used to determine if a field in a fact matches the pattern node. For example, the expression may indicate that the field of the fact must either be the constant red or the constant green.

**ARGUMENTS:** A pointer to the current field in the fact being tested, an expression to be evaluated, and a pointer to the pattern node with which the expression is associated.

**RETURNS:** Boolean value. TRUE if the expression evaluates to TRUE and FALSE if the expression evaluates to FALSE.

**OTHER NOTES:** Implemented as a separate function from the general purpose function EvaluateExpression to allow a fast evaluation of common pattern network expressions. Functions evaluated are constant and notconstant (which determine respectively whether a field is equal or not equal to a particular constant), eqfield and neqfield (which determine respectively whether two fields in the same fact are either equal or not equal), and the functions and and or. All other functions are evaluated using the function EvaluateExpression.

### TraceErrorToPattern

**PURPOSE:** Prints an error message when a error occurs as the result of evaluating an expression in the pattern network. Used to indicate which patterns caused the problem (e.g. 1st pattern of a rule, 2nd pattern of a rule, etc). Calls the function TraceErrorToRule to further isolate the error.
ARGUMENTS: A pointer to the pattern node being matched against when the error occurred.
The Retract Module (retract.c) handles the major functionality of updating the join network when a fact has been retracted from the knowledge base. The algorithm for retracting a fact from the knowledge base is described as follows.

As pattern matching occurs, information is stored with each fact to indicate which patterns have matched a given fact. Thus, it is not necessary to perform pattern matching during retraction of a fact as all matched patterns are already known. Given the list of patterns matched by a fact to be retracted, the Retract Module will loop through the list of patterns and perform appropriate retraction operations for that pattern. Each pattern matched has an associated “end of pattern” node which is connected to a series of joins. Joins entered from a not pattern CE will use different algorithms for handling a fact retraction than joins entered from a pattern CE. In addition, retraction from the join associated with the first CE of a rule is handled differently. Each algorithm will be discussed in greater detail in the following paragraphs. Once retraction has been performed for all joins associated with a specific pattern node, the alpha memory partial match corresponding to the retracted fact can be removed from the alpha memory of the pattern node.

The first algorithm for retraction occurs when the join being entered has a pattern CE associated with its RHS. If the join was entered from the RHS, the partial match containing the retracted fact is known to exist in the alpha memory associated with this join (and will be removed later). Otherwise, the beta memory of the join is searched to find and delete all partial matches containing the fact. If the fact is not found in the join, then retraction for this portion of the join network is complete. Otherwise, this algorithm is used to recursively retract the fact from all child joins of the current join. Note that a level of recursion is removed from this algorithm by performing retraction recursively for all but one of the child joins attached to the current join. This last join is handled non-recursively within the main loop of the algorithm. Thus, in the event that every join has a single child join, this algorithm will stay within a relatively tight loop avoiding recursion.

The second algorithm for retraction occurs when the join is associated with a not pattern CE that is the first CE of a rule. First, the number of occurrences of the fact in partial matches found in the alpha memory associated with this join is determined. These occurrences represent the facts preventing the not pattern CE from being satisfied. The id slot of the join (which in the top-most join represents the number of facts preventing the not pattern CE from being satisfied) can then be decremented by the number of fact occurrences found in the alpha memory. If, after decrementing, the join id is not zero, then the not pattern CE still has facts preventing it from being satisfied and the retraction process for this join is completed. Otherwise, the not pattern CE has been satisfied. If the secondary join expression (for test CEs following a not pattern CE) associated with the join is also satisfied, a pseudo-fact partial match for the not pattern CE can be created and sent to all the child joins connected to this join. The newly created partial match is indirectly sent to the others joins by using the DriveRetractions function.

The third algorithm for retraction occurs when the join is associated with a not pattern CE that is not the first CE of a rule. This algorithm is similar to the second algorithm. The major exception is that the count for the number of facts preventing the join from being satisfied is stored in the beta memory partial matches. This algorithm uses a double loop, looping through the alpha memory in the outer loop and the beta...
memory in the inner loop. Within the inner loop, if the alpha memory partial match being tested corresponds to the fact being retracted, the primary join expression is evaluated for the current alpha and beta memory partial matches. If the expression evaluates to TRUE (or was non-existent), the alpha memory partial match conflicted with the beta memory partial match. The count of conflicting facts in the beta memory partial match can be decremented by one. If the beta memory count reaches zero and the second join expression associated with the join evaluates to TRUE, a new partial match consisting of a pseudo-fact partial match and the beta memory partial match combined is created and sent to all the child joins connected to this join. The newly created partial match is indirectly sent to the others joins by using the DriveRetractions function.

GLOBAL VARIABLES

- **GarbageAlphaMatches**
  
PURPOSE: Maintains a list of data structures which represent the pseudo-facts that matched not CE if no real facts matched the CE. Like facts, these data structures are not thrown away during rule execution, since the rule may still refer to the data structure.

- **GarbagePartialMatches**
  
PURPOSE: Maintains a list of partial matches associated with a not CE or an alpha memory. Like facts, these data structures are not thrown away during rule execution, since the rule may still refer to the data structure.

INTERNAL VARIABLES

- **DriveRetractionList**
  
PURPOSE: Maintains a list of partial matches that are to be driven through the join network as the result of a not CE being satisfied by a retraction.

GLOBAL FUNCTIONS

- **DeletePartialMatches**
  
PURPOSE: Searches through a list of partial matches and removes any partial match that contains the specified fact-index.

ARGUMENTS: A fact-index, a list of partial matches, and a pointer to an integer flag which indicates if any partial matches were deleted, an integer indicating the position in the partial
match to be searched for the fact-index, and an integer flag indicating whether the list of partial matches is associated with an alpha or beta memory.

RETURNS: Returns the modified list of partial matches. The integer flag is set to TRUE if any partial matches were deleted. Otherwise, it is set to FALSE.

FlushGarbagePartialMatches

PURPOSE: Returns partial matches and associated structures that were removed as part of a retraction. It is necessary to postpone returning these structures to memory because RHS actions retrieve their variable bindings directly from the fact data structure through the alpha memory bindings.

ARGUMENTS: None. Makes use of the GarbageAlphaMatches and GarbagePartialMatches variables.

NetworkRetract

PURPOSE: Coordinates the retraction of a fact from the join and pattern networks.

ARGUMENTS: A list of the patterns that the fact matched and the fact-index of the fact being retracted.

OTHER NOTES: See algorithm above.

PosEntryRetract

PURPOSE: Handles removing partial matches from a join and all child joins that contain a specified fact. Used for joins that's RHS entry is associated with a pattern CE.

ARGUMENTS: Direction from which the join was entered, a pointer to the join, the fact-index of the fact to be removed from the join, and the position where the fact-index should be found in the partial matches.

ReturnPartialMatch

PURPOSE: Returns a partial match and its associated data structures to the CLIPS memory manager. If the partial match is busy (i.e. it is currently in use) it is placed on the list of GarbagePartialMatches.

ARGUMENTS: A pointer to a partial match.
INTERNAL FUNCTIONS

DriveRetractions

PURPOSE: Drives partial matches generated by the retraction of a fact through the join network.

ARGUMENTS: None. Uses the DriveRetractionList to determine new partial matches to be driven through the join network.

OTHER NOTES: The retraction of a fact can generate new partial matches that must be driven through the join network if that fact matched a not CE. However, such a fact may also match pattern CEs in the same rule. Therefore, to prevent partial matches being generated from facts that are to be retracted, propagation of new partial matches through the join network is delayed until all partial matches containing a fact to be retracted have been removed from the join and pattern networks.

NegEntryRetract

PURPOSE: Handles retractions from the RHS for joins associated with a not CE that are not associated with the first pattern of a rule.

ARGUMENTS: A pointer to a join and the fact ID of the fact to be retracted.

ReturnMarkers

PURPOSE: Returns the list of data structures associated with an alpha memory partial match that indicate how multifield variables matched the pattern associated with the partial match.

ARGUMENTS: A list of data structures.

TopNegJoinRetract

PURPOSE: Handles retractions from the RHS for top-level joins (i.e. joins associated with the first pattern of a rule) associated with a not CE.

ARGUMENTS: A pointer to a top-level join and the fact-index of the fact to be retracted.
The Rete Utility Module (reteutil.c) contains a number of functions that are useful to other modules for implementing the Rete algorithm.

GLOBAL VARIABLES

**GlobalLHSBinds**

PURPOSE: A pointer to the partial match currently being examined on the LHS of a join as part of the pattern matching process. Also used to point to the partial match associated with the activation for the currently executing rule. This variable is used by a number of functions (such as the `get_var` function) to extract a value from the LHS of a rule for use in a function call.

**GlobalRHSBinds**

PURPOSE: A pointer to the partial match currently being examined on the RHS of a join as part of the pattern matching process. This variable is used by a number of functions (such as the `get_var` function) to extract a value from the LHS of a rule for use in a function call.

INTERNAL VARIABLES

**PseudoFactIndex**

PURPOSE: Contains the next fact index to be used when creating a “pseudo” fact. Pseudo facts are used in the pattern matching process to indicate that a `not` conditional element has no facts matching it. Pseudo facts have a fact index which is less than zero.

GLOBAL FUNCTIONS

**AddSingleMatch**

PURPOSE: Adds an additional alpha match to a partial match.

ARGUMENTS: A pointer to a partial match and a pointer to an alpha match from the pattern network.

RETURNS: A pointer to a new partial match which consists of the single alpha match appended to the first partial match (the original partial match is unaffected).
AdjustFieldPosition

PURPOSE: Given the number of fields each multifield variable or wildcard in a pattern has actually matched, determines the actual index of a variable within a pattern in the matching fact. For example, given the pattern (data $?x c $?y ?z) and the fact (data a b c d e f x), the actual index in the fact for the 5th item in the pattern (the variable ?z) would be 8 since $?x binds to 2 fields and $?y binds to 3 fields.

ARGUMENTS: A pointer to a list of data structures describing the fields that each multifield variable or wildcard has matched, an integer indicating the position of the variable within the pattern, and a pointer to an integer which stores the extent of the variable (the number of fields the variable has matched—zero or greater for multifield variables).

RETURNS: The index of the variable within the fact matched by the pattern. The extent of the variable (if it is a multifield) is also stored in one of the calling arguments (which should be initialized either to 1 if there is no need to distinguish between the extent of single field and multifield variables or -1 if there is a need).

ClearPatternMatches

PURPOSE: Removes all links between a pattern and the facts that have matched that pattern.

ARGUMENTS: A pointer to a pattern node (which should be an end of pattern pattern node).

CopyPartialMatch

PURPOSE: Copies a partial matches.

ARGUMENTS: A pointer to a partial matches to be copied.

RETURNS: A pointer to a copy of the partial match.

FindFactInPartialMatch

PURPOSE: Searches for a specified fact index in a partial match.

ARGUMENTS: Fact index for which to search and a pointer to a partial match.

RETURNS: Boolean value. TRUE if the fact index is found; otherwise FALSE.
**FlushAlphaBetaMemory**

**PURPOSE:** Returns all partial matches in a list of partial matches either directly to the pool of free memory or to the list of **GarbagePartialMatches**.

**ARGUMENTS:** A pointer to a list of partial matches.

**IncrementPseudoFactIndex**

**PURPOSE:** Decrements the current value of **PseudoFactIndex** and returns the previous value.

**RETURNS:** The current value of the global variable **PseudoFactIndex**.

**MarkRuleNetwork**

**PURPOSE:** Marks each node in the pattern and join networks with a boolean value (typically indicating whether an action has been taken for that node). This mark value is used by binary save and the construct compiler.

**ARGUMENTS:** The boolean value that the nodes are to be marked with (TRUE or FALSE).

**RETURNS:** No return value. The boolean value is stored in the marked slot of the pattern and join nodes.

**MergePartialMatches**

**PURPOSE:** Combines two partial matches into a single partial match.

**ARGUMENTS:** A pointer to a partial match and another pointer to a partial match.

**RETURNS:** A pointer to a new partial match which consists of the second partial matched appended to the first partial match (the original partial matches are unaffected).

**NewPseudoFactPartialMatch**

**PURPOSE:** Creates a partial match consisting of a pseudo fact index associated with a not CE.

**RETURNS:** A partial match consisting of the pseudo fact index. The value of **PseudoFactIndex** is also decremented by this function.
**GetNumericArgument**

**PURPOSE:** Directly evaluates a numeric expression under certain conditions.

**ARGUMENTS:** A pointer to an expression to evaluate, the name of the function being executed, a pointer to a DATA_OBJECT structure in which to store the result of the evaluation, a boolean flag indicating whether integer results should be converting to floating point, and an integer value representing the position of the expression within the argument list of the function being executed.

**RETURNS:** Boolean value. TRUE if the result of the expression evaluation was a number, otherwise FALSE. The value of the number is also stored in the DATA_OBJECT structure.

**OTHER NOTES:** Used to provide fast evaluate of arguments to basic arithmetic functions. If the argument is a number or a variable, it is evaluated immediately, otherwise, **EvaluateExpression** is used.

---

**PrimeJoin**

**PURPOSE:** Updates a join in a rule for an incremental reset. Joins are updated by “priming” them only if the join is shared with other rules that have already been incrementally reset. A join for a new rule will be updated if it is marked for initialization and either its parent join or its associated entry pattern node has not been marked for initialization.

**ARGUMENTS:** A pointer to a join.

---

**PrintPartialMatch**

**PURPOSE:** Prints out a list of fact indices associated with a partial match or rule instantiation.

**ARGUMENTS:** A logical name to which output is sent and a pointer to a partial match.

---

**ResetDeployedRuleImage**

**PURPOSE:** Incrementally resets a runtime image created using the construct compiler. This function is called by the **InitCImage** function associated with the runtime image.
**ResetNotedJoin**

PURPOSE: Determines if a given join is associated with a *not* CE that is the first pattern of a rule and also whether it should be initialized. If the join needs to be initialized a partial match for the join is created. This function is called as part of an incremental reset.

ARGUMENTS: A pointer to a join node.

**ResetNotedPatterns**

PURPOSE: Searches for and generates partial matches for *not* CEs that are the first pattern of a rule. This function is called as part of an incremental reset for runtime images create with the construct compiler and is called by the function `ResetDeployedRuleImage`.

ARGUMENTS: A pointer to a pattern node in the pattern network. This function uses recursion to traverse the pattern network. This first call to this function should pass the root node of the pattern network as its argument.

**SetPseudoFactIndex**

PURPOSE: Sets the value of the global variable `PseudoFactIndex`.

ARGUMENTS: New integer starting value for pseudo fact indices (which must be less than zero).

**TagRuleNetwork**

PURPOSE: Assigns a unique integer value to each pattern node in the pattern network and each join node in the join network. This ID value is used by binary save and the construct compiler.

ARGUMENTS: A pointer to an integer value containing the number of pattern nodes encountered and the number of join nodes encountered. The value of these integer variables should be set to zero before this function is called.

RETURNS: No return value. The pointers passed as arguments have their values respectively set to the number of pattern nodes and join nodes found in the rule network.

C IMPLEMENTATION: The integer ID value is stored in the bsaveID slot of the pattern and join nodes.
INTERNAL FUNCTIONS

None.
The Logical Dependencies Module (lgcldpnd.c) provides the support routines necessary for the implementation of the logical conditional element. A fact asserted by a rule without logical CEs in the rule's LHS is unconditionally supported. A fact that is unconditionally supported can only be explicitly retracted (i.e., it cannot be retracted as the direct result of retracting another fact). Facts asserted by deffacts, from the top-level command prompt, or as the result of actions occurring outside the scope of a executing rule containing logical CEs are also unconditionally supported. A fact asserted by a rule with logical CEs in the rule's LHS is logically supported by that rule. The group of facts contained within the logical CE provide the logical support for the asserted fact.

Since the logical CEs must appear as the first patterns on the LHS of a rule and there can be no gaps between logical CEs, there exists a partial match for the rule containing all of the facts providing logical support in the beta memory of one of the rule's joins. Logical dependencies are implemented by maintaining two types of links. First, links are created between a partial match and each fact which receives logical support from that partial match. Second, reverse links are created between facts and the partial matches which provide logical support to them.

When a defrule is parsed that contains logical CEs, the location of the join that will contain the partial matches providing logical support is computed. A pointer to this join is saved as part of the defrule's data structure. When the rule is then executed, the pointer to the join is stored in the global variable TheLogicalJoin. Assertions that occur from the RHS of the executing rule can then create the appropriate links between the partial match stored in the join referenced by TheLogicalJoin and the facts to which it provides logical support. When partial matches are removed from the beta memories of a join (either as the result of a retract or an assert), then the links between that partial match and the facts it supports are updated. If a fact loses all of its logical support, then it is automatically retracted.

The following two rules will be used to illustrate the links used to support logical dependencies.

```
(defrule Example-1
 (logical (a)
 (b))
 (d)
 =>
 (assert (e) (f)))

(defrule Example-2
 (logical (b)
 (c))
 (d)
 =>
 (assert (f) (g)))
```

Assuming the rules have already been loaded, the following commands will execute the rules creating three new facts which are logically supported.

```
CLIPS> (reset)
```
The following commands illustrate the logical dependency links between the facts.

The logical support links between the partial matches and the facts dependent upon the partial match are shown in the following diagram. The facts (e) and (f) asserted by rule Example-1 are logically dependent upon the partial match containing facts (a) and (b). Similarly, the facts (f) and (g) asserted by rule Example-2 are logically dependent upon the partial match containing facts (b) and (c). Note that each partial match supports two different facts.
The logical support links between the facts and the partial matches from which they receive logical are shown in the following diagram. Fact (e) is logically supported by the partial match containing facts (a) and (b) from rule Example-1. Fact (f) is logically supported by the partial match containing facts (a) and (b) from rule Example-1 and the partial match containing facts (b) and (c) from rule Example-2. Fact (g) is logically supported by the partial match containing facts (b) and (c) from rule Example-2. Note that fact (f) receives logical support from two different partial matches.
GLOBAL VARIABLES

None.

INTERNAL VARIABLES

| DependencyList |

PURPOSE: A list of pointers to facts that are to be removed because all of their logical support has been removed.
GLOBAL FUNCTIONS

AddLogicalDependencies

PURPOSE: Adds the logical dependency links between a fact and the partial match which logically supports that fact. If a fact is unconditionally asserted (i.e. the global variable TheLogicalJoin is NULL), then existing logical support for the fact is no longer needed and it is removed. If a fact is already unconditionally supported and the fact is conditionally asserted (i.e. the global variable TheLogicalJoin is not NULL), then the logical support is ignored. Otherwise, the partial match is linked to the fact and the fact is linked to the partial match.

ARGUMENTS: A pointer to a fact and a boolean flag indicating if the fact already exists. If a fact already exists, it just receives additional logical support.

RETURNS: Boolean value. TRUE if the fact should be asserted, otherwise FALSE. A value of FALSE is returned when the logical support for a fact is removed before the fact is asserted (e.g. by retracting a fact contained in a logical CE of a rule before asserting the fact dependent on the partial match associated with the logical CE).

AddToDependencyList

PURPOSE: Removes the dependency links between a partial match and the facts it logically supports. Also removes the associated links from the facts which point back to the partial match by calling DetachAssociatedFactDependencies. If a fact has all of its logical support removed as a result of this procedure, the dependency link from the partial match is added to the DependencyList so that the fact will be retracted as a result of losing its logical support.

ARGUMENTS: A pointer to a partial match.

ForceLogicalRetractions

PURPOSE: Retracts the first fact found on the DependencyList by calling RetractFact. This retraction will then trigger the retract of the remaining facts on the DependencyList since RetractFact will call GetNextLogicalRetraction. This function is called by AddFact after a new fact has been processed because the addition of a new fact may cause partial matches associated with a not conditional element to be removed. RetractFact does not call this function since it
automatically processes the retraction of facts that lose their logical support.

**GetNextLogicalRetraction**

**PURPOSE:** Returns the next fact on the **DependencyList** that is to be retracted because all of its logical support has been removed.

**RETURNS:** A pointer to a fact. The **DependencyList** is also modified (the first item on the list is removed).

**ListDependencies**

**PURPOSE:** Lists the partial matches from a specified fact receives logical support.

**ARGUMENTS:** A pointer to a fact.

**ListDependents**

**PURPOSE:** Lists all facts which receive logical support from a specified fact.

**ARGUMENTS:** A pointer to a fact.

**RemoveFactDependencies**

**PURPOSE:** Removes all logical support links from a fact that point to any partial matches. Also removes the associated links from the partial matches which point back to the fact by calling **DetachAssociatedPMDependencies**.

**RemovePMDependencies**

**PURPOSE:** Removes all logical support links from a partial match that point to any facts. Also removes the associated links from the facts which point back to the partial match by calling **DetachAssociatedFactDependencies**.

**ARGUMENTS:** A pointer to a fact.
LOCAL FUNCTIONS

DetachAssociatedFactDependencies

PURPOSE: Removes all logical support links from a fact that point to a specified partial match. Does not remove links which may point back to the fact from the partial match.

ARGUMENTS: A pointer to a fact and a pointer to a partial match.

DetachAssociatedPMDependencies

PURPOSE: Removes all logical support links from a partial match that point to a specified fact. Does not remove links which may point back to the partial match from the fact.

ARGUMENTS: A pointer to a partial match and a pointer to a fact.

FindLogicalBind

PURPOSE: Finds the partial match associated with the logical CE which will provide logical support for a fact asserted from the currently executing rule. The function is called by AddLogicalDependencies when creating logical support links between the facts and supporting partial matches. It compares each partial match found at a specified join to the partial match associated with a rule activation until it finds the partial match that generated the rule activation.

ARGUMENTS: A pointer to a join data structure and a partial match. Called by the function AddLogicalDependencies with the values TheLogicalJoin and GlobalLHSBinds.

RETURNS: A pointer to a partial match (or NULL if the appropriate partial match could not be found).
The Defrule Manager Module (defrule.c) contains a set of functions which initialize and provide high level support for the defrule construct.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

DeletedRuleHadBreakpoint
PURPOSE: Boolean flag. If TRUE, the last rule deleted had a breakpoint set. This flag is used to restore the breakpoint status of a rule that is redefined.

DeletionsAllowed
PURPOSE: Boolean flag. If TRUE, indicates that rules can be deleted. Rules cannot be deleted during certain operations such as assertion and retraction of facts.

LastDefrule
PURPOSE: A pointer to the last defrule in the ListOfDefrules.

ListOfDefrules
PURPOSE: A linked list of all the currently defined defrules.

WatchRules
PURPOSE: Boolean flag. If TRUE, indicates that rule firings should be displayed.

GLOBAL FUNCTIONS

ClearDefrules
PURPOSE: Defrule construct clear function. Deletes all defrules.

DeleteDefrule
PURPOSE: Deletes a defrule from the ListOfDefrules.
ARGUMENTS: A pointer to the defrule to be deleted.
DeleteNamedDefrule

PURPOSE: Deletes a named defrule from the ListOfDefrules.

ARGUMENTS: The name of the defrule to be deleted.

RETURNS: Boolean value. TRUE if the defrule was found and deleted, otherwise FALSE.

EvaluateSalience

PURPOSE: Returns the salience value of the specified defrule. If salience evaluation is currently set to when-defined, then the current value of the rule's salience is returned. Otherwise the salience expression associated with the rule is reevaluated, the value is stored as the rule's current salience, and it is then returned.

ARGUMENTS: A pointer to a defrule data structure.

RETURNS: The current salience value of the rule. The slot value for the defrule's current salience value is also changed if needed.

FindDefrule

PURPOSE: Finds a named defrule in the ListOfDefrules.

ARGUMENTS: The name of the defrule to be found.

RETURNS: A pointer to the defrule if found, otherwise NULL.

GetDefruleName

PURPOSE: Returns the name of a defrule.

ARGUMENTS: A pointer to a defrule.

RETURNS: String name of the defrule.

GetDefrulePPForm

PURPOSE: Returns the pretty print representation of a defrule.

ARGUMENTS: A pointer to a defrule.

RETURNS: The string pretty print representation of the defrule.
GetDisjunctIndex

PURPOSE: Returns the disjunct index of a defrule. Disjuncts are created when *or* conditional elements are used on the LHS of a rule. Each disjunct acts as a separate rule and handles one permutation created by the *or* CEs on the LHS of the rule.

ARGUMENTS: A pointer to a defrule data structure (the disjunct that's index is being sought).

RETURNS: An integer value. If the disjunct cannot be found -1 is returned. Otherwise an integer value ranging from zero to one less than the number of disjuncts for the rule.

GetIndexedDefrule

PURPOSE: Allows access to the *ListOfDefrules* by returning a pointer to nth defrule in the *ListOfDefrules*.

ARGUMENTS: Integer index of the rule desired in the *ListOfDefrules*.

RETURNS: A pointer to the specified defrule. If the index is greater than the number of rules in the *ListOfDefrules*, a NULL pointer is returned.

GetNextDefrule

PURPOSE: Allows access to the *ListOfDefrules*.

ARGUMENTS: A pointer to a deffacts in the *ListOfDefrules*.

RETURNS: If passed a NULL pointer, returns the first defrule in the *ListOfDefrules*. Otherwise, returns the next defrule following the defrule passed as an argument.

GetRuleDeletions

PURPOSE: Returns the current value of the *RuleDeletions* flag.

RETURNS: A boolean value.

GetRulesWatch

PURPOSE: Returns the current value of the *WatchRules* flag.

RETURNS: A boolean value.
### InitializeDefrules

**PURPOSE:** Initializes the defrule construct. Adds the rules watch item, adds **reset**, **clear**, and **save** functions for defrules, and calls the functions **DefruleCommands** and **InitializeEngine** to define other defrule related commands and features.

### IsDefruleDeletable

**PURPOSE:** Indicates whether a defrule can be deleted.

**ARGUMENTS:** A pointer to a defrule.

**RETURNS:** Boolean value. TRUE if the defrule can be deleted, otherwise FALSE.

### ListMatches

**PURPOSE:** Prints all of the partial matches and activations for a specified defrule.

**ARGUMENTS:** A pointer to the defrule for which matches are to be listed.

**RETURNS:** Boolean value. TRUE if the matches were listed, otherwise FALSE.

### RefreshDefrule

**PURPOSE:** Refreshes a defrule. Activations of the rule that have already been fired are added to the agenda.

**ARGUMENTS:** A pointer to the defrule to be refreshed.

**RETURNS:** Boolean value. TRUE if the defrule was successfully refreshed, otherwise FALSE.

### ReturnDefrule

**PURPOSE:** Returns a defrule data structure to the CLIPS memory manager.

**ARGUMENTS:** A pointer to a defrule data structure.

**RETURNS:** Boolean value. TRUE if the data structure was successfully returned to the CLIPS memory manager, otherwise FALSE.
SalienceInformationError

PURPOSE: Prints an informational message indicating which rule's salience declaration caused an error when a salience value was being evaluated.

ARGUMENTS: The name of a rule.

SetListOfDefrules

PURPOSE: Sets the ListOfDefrules to the specified value. Normally used when initializing a run-time module or when loading a binary file to install the ListOfDefrules.

ARGUMENTS: A pointer to a linked list of defrules.

SetRuleDeletions

PURPOSE: Sets the current value of the RuleDeletions flag.

ARGUMENTS: A boolean value (the new value of the flag).

RETURNS: A boolean value (the old value of the flag).

LOCAL FUNCTIONS

AddDefrule

PURPOSE: Adds a defrule to the ListOfDefrules and updates the value of the variable LastDefrule.

ARGUMENTS: A pointer to the defrule data structure.

AddTerminatorJoin

PURPOSE: Creates the final join for a rule which contains the partial matches for the activations of the rule.

RETURNS: A pointer to a join node data structure.

CheckForPrimableJoins

PURPOSE: Updates the joins of a rule for an incremental reset if portions of that rule are shared with other rules that have already been incrementally reset. A join for a new rule will be updated if it is marked for initialization and either its parent join or its associated entry pattern node has not been
marked for initialization. The function **PrimeJoin** is used to update joins which meet these criteria.

**ARGUMENTS:** A pointer to a defrule data structure.

---

### IncrementalReset

**PURPOSE:** Incrementally resets the specified rule. First, any rules containing **not** CEs as the first CE are checked to see if that CE is satisfied. Second, if a rule shares patterns or joins with other rules, it may be necessary to update the join network based on existing partial matches. Third, existing facts are driven through the new portions of the pattern and join networks.

**ARGUMENTS:** A pointer to a defrule data structure.

---

### MarkNetworkForIncrementalReset

**PURPOSE:** Used to set the initialization flags of the pattern and join nodes for a specified rule before and after an incremental reset is performed.

**ARGUMENTS:** A pointer to a defrule data structure and a boolean value to be assigned to the pattern and join nodes.

**OTHER NOTES:** The assignment of the initialization value is partially dependent upon certain joins already having their initialization flags set to TRUE.

---

### ParseDefrule

**PURPOSE:** Coordinates all actions necessary for the construction of a defrule into the current environment. Called to parse a defrule construct.

**ARGUMENTS:** Logical name from which defrule input is read.

**RETURNS:** Boolean value. TRUE if an error occurred while parsing the defrule, otherwise FALSE.

**OTHER NOTES:** Makes use of parsing functions from other modules such as **GetConstructNameAndComment**, **ParseRuleLHS**, and **ParseRuleRHS**.

---

### RemoveRuleNetwork

**PURPOSE:** Removes the pattern and join nodes for a specified rule from the pattern and join networks.
ARGUMENTS: A pointer to a defrule data structure.

OTHER NOTES: Uses the function **DetachJoins** to remove the rule from the pattern and join networks.

---

### ReplaceExpressionVariables

**PURPOSE:** Replaces all symbolic references to variables (local and global) found in an expression on the RHS of a rule with expressions containing function calls to retrieve the variable's value. Makes the final modifications necessary for handling the **modify** and **duplicate** commands.

**ARGUMENTS:** A pointer to an expression and a pointer to an integer for storing an error flag.

**RETURNS:** Nothing. If an error occurs the value of the error flag passed as a pointer is set to TRUE.

**OTHER NOTES:** Makes use the functions **ReplaceRHSVariable**, **ReplaceGlobalVariable**, and **UpdateModifyDuplicate** to update the expression.

---

### RememberJoinsForRule

**PURPOSE:** Stores information for each rule of how it is attached to the pattern and join networks. This information is stored in a linked list attached to the defrule data structure for the rule.

**ARGUMENTS:** A pointer to the rule's terminator join, a pointer to the defrule data structure for the rule, and the depth index of the logical join for the rule.

---

### ReplaceRHSVariable

**PURPOSE:** Replaces a symbolic reference to single- or multifield local variable found in an expression on the RHS of a rule with an expression containing a function call to retrieve the variable's value.

**ARGUMENTS:** A pointer to an expression.

**RETURNS:** Boolean value. TRUE if the variable reference was successfully replaced, otherwise FALSE.

---

### ResetDefrules

**PURPOSE:** Defrule construct reset function. Reinitializes rules that have a **not** conditional element as their first conditional element.
SaveDefrules

PURPOSE: Defrule. construct save function. Pretty prints all defrules to the given logical name.

ARGUMENTS: A logical name to which output is sent.
Defrule Deployment Module

The Defrule Deployment Module (drulebin.c) provides the functionality for implementing the `bload`, `bsave`, and `constructs-to-c` functions for the defrule construct.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DefruleArray</td>
<td>A pointer to an array of defrule data structures loaded using the <code>bload</code> command.</td>
</tr>
<tr>
<td>JoinArray</td>
<td>A pointer to an array of join node data structures loaded using the <code>bload</code> command.</td>
</tr>
<tr>
<td>NumberOfDefrules</td>
<td>An integer count of the number of defrule data structures in the <code>DefruleArray</code>.</td>
</tr>
<tr>
<td>NumberOfJoins</td>
<td>An integer count of the number of join node data structures in the <code>JoinArray</code>.</td>
</tr>
<tr>
<td>NumberOfPatternPointers</td>
<td>An integer count of the number of pattern pointer data structures in the <code>PatPtrArray</code>.</td>
</tr>
<tr>
<td>NumberOfPatterns</td>
<td>An integer count of the number of pattern node data structures in the <code>PatternArray</code>.</td>
</tr>
<tr>
<td>PatPtrArray</td>
<td>A pointer to an array of pattern pointer data structures loaded using the <code>bload</code> command.</td>
</tr>
</tbody>
</table>
PatternArray

PURPOSE: A pointer to an array of pattern node data structures loaded using the `bload` command.

GLOBAL FUNCTIONS

DefruleBinarySetup

PURPOSE: Initializes the `bload`, `bsave`, and `constructs-to-c` functions for the defrule construct.

INTERNAL FUNCTIONS

Defrule Bload/Bsave Functions

PURPOSE: A set of functions used by the `bload` and `bsave` commands to process the defrule construct. These functions are made available to the `bload` and `bsave` commands by calling the function `AddBinaryItem`.

Defrule Constructs-To-C Functions

PURPOSE: A set of functions used by the `constructs-to-c` command to process the defrule construct. These functions are made available to the `constructs-to-c` command by calling the function `AddCodeGeneratorItem`. 
The Defrule Commands Module (rulecom.c) provides a number of commands for manipulating and examining defrules. The commands provided are run, undefrule, refresh, halt, rules, ppdefrule, get-incremental-reset, set-incremental-reset, set-break, remove-break, show-breaks, matches, agenda, get-strategy, set-strategy, get-salience-evaluation, set-salience-evaluation, and refresh-agenda.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

None.

GLOBAL FUNCTIONS

**DefruleCommands**

**PURPOSE:** Makes appropriate DefineFunction calls to notify CLIPS of functions defined in this module.

INTERNAL FUNCTIONS

**Defrule Commands**

**PURPOSE:** A series of commands which define the defrule commands listed above. See the Basic Programming Guide for more detail on individual functions.

**OTHER NOTES:** Some functionality for these commands is provided in other modules.
Deftemplate Commands Module

The Deftemplate Commands Module (deftmcom.c) manages commands associated with the deftemplate construct. These commands include `undeftemplate`, `ppdeftemplate`, `list-deftemplates`, `modify`, `duplicate`, `set-dynamic-deftemplate-checking`, and `get-dynamic-deftemplate-checking`. Extensions for the `save`, `clear`, `bload`, `bsave`, and `constructs-to-c` commands are defined by this module. Several support functions for deftemplates are also provided by this module. For a description of the deftemplate construct, see the CLIPS Reference Manual. The deftemplate construct capability can be removed by using the appropriate compile flag in the setup header file.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

<table>
<thead>
<tr>
<th>DeftemplateArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: A pointer to an array of deftemplate data structures loaded using the <code>bload</code> command.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DeftemplateHashTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Stores all deftemplates used by CLIPS.</td>
</tr>
<tr>
<td>C IMPLEMENTATION: Implemented as an array. Each position in the array corresponds to a list of deftemplate entries. Collisions are resolved by adding the deftemplate entry to the list of entries.</td>
</tr>
<tr>
<td>OTHER NOTES: Information about deftemplates is also stored in the <code>ListOfDeftemplates</code>. This table is primarily used to quickly locate a deftemplate.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DynamicDeftemplateChecking</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Boolean flag. If TRUE, indicates that dynamic deftemplate constraint checking is performed for newly asserted deftemplate facts. If this flag is FALSE, then no checking is performed when a fact is asserted.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LastDeftemplate</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: A pointer to the last deftemplate in the <code>ListOfDeftemplates</code>.</td>
</tr>
</tbody>
</table>
**ListOfDeftemplates**

**PURPOSE:** A linked list of all the currently defined deftemplates.

---

**NumberOfDeftemplates**

**PURPOSE:** An integer count of the number of deftemplate data structures in the DeftemplateArray.

---

**NumberOfTemplateSlots**

**PURPOSE:** An integer count of the number of slot data structures in the SlotArray.

---

**SlotArray**

**PURPOSE:** A pointer to an array of slot data structures loaded using the **bload** command.

---

**GLOBAL FUNCTIONS**

---

**AddDeftemplate**

**PURPOSE:** Adds a deftemplate to the **ListOfDeftemplates**, then installs the deftemplate using **InstallDeftemplate**.

**ARGUMENTS:** A pointer to a deftemplate data structure.

---

**CheckSlotAllowedValues**

**PURPOSE:** Determines if a primitive data type satisfies the allowed-... attributes of a slot.

**ARGUMENTS:** The type of the primitive data type, the value of the primitive data type, and a pointer to a deftemplate slot data structure.

**RETURNS:** Boolean value. FALSE if the allowed-... attribute is violated, otherwise TRUE.

---

**CheckSlotRange**

**PURPOSE:** Determines if a primitive data type satisfies the range attribute of a slot.

**ARGUMENTS:** The type of the primitive data type, the value of the primitive data type, and a pointer to a deftemplate slot data structure.
RETURNS: Boolean value. FALSE if the range attribute is violated, otherwise TRUE.

CheckSlotType

PURPOSE: Determines if a primitive data type satisfies the type attribute of a slot.

ARGUMENTS: The type of the primitive data type and a pointer to a deftemplate slot data structure.

RETURNS: Boolean value. FALSE if the type attribute is violated, otherwise TRUE.

ClearDeftemplates

PURPOSE: Deftemplates construct clear function. Removes all deftemplates from the ListOfDeftemplates.

DeleteDeftemplate

PURPOSE: Deletes a deftemplate from the ListOfDeftemplates.

ARGUMENTS: A pointer to the deftemplate to be deleted.

RETURNS: Boolean value. TRUE if the deftemplate was found and deleted, otherwise FALSE.

DeleteNamedDeftemplate

PURPOSE: Deletes a named deftemplate from the ListOfDeftemplates.

ARGUMENTS: The name of deftemplate to be deleted.

RETURNS: Boolean value. TRUE if the deftemplate was found and deleted, otherwise FALSE.

DuplicateCommand

PURPOSE: Implements the duplicate command. Calls the function DuplicateModifyCommand with a value of FALSE to execute the command.

FindDeftemplate

PURPOSE: Finds a named deftemplate in the ListOfDeftemplates.

ARGUMENTS: The name of deftemplate to be found.
RETURNS: A pointer to the deftemplate if found, otherwise NULL.

**FindSlot**

PURPOSE: Finds the specified slot in a deftemplate.

ARGUMENTS: The name of the slot to be found and a pointer to a deftemplate data structure.

RETURNS: A pointer to a slot data structure if the slot is valid for the specified deftemplate, otherwise NULL.

**FindSlotItem**

PURPOSE: Given a list of slot assignments, finds the assignment which matches a specified slot.

ARGUMENTS: A pointer to a slot data structure and a list of slot assignments.

RETURNS: The slot assignment matching the specified slot data structure, or NULL if there was no match.

**FindSlotPosition**

PURPOSE: Returns the integer position of the specified slot for facts using a specified deftemplate. Single-field slots are ordered in the same position that they were defined and the multifield slot is positioned after all other slots.

ARGUMENTS: A pointer to a deftemplate data structure and the name of a slot.

RETURNS: An integer index ranging from 1 to the number of slots in the deftemplate. Zero is returned is the slot is not associated with the specified deftemplate.

**GetDeftemplateName**

PURPOSE: Returns the name of a deftemplate.

ARGUMENTS: A pointer to a deftemplate.

RETURNS: String name of the deftemplate.

**GetDeftemplatePPForm**

PURPOSE: Returns the pretty print representation of a deftemplate.
ARGUMENTS: A pointer to a deftemplate.

RETURNS: The string pretty print representation of the deftemplate.

**GDDCommand**

PURPOSE: Implements the `get-dynamic-deftemplate-checking` command.

**GetDynamicDeftemplateChecking**

PURPOSE: Returns the current value of the `DynamicDeftemplateChecking` flag.

RETURNS: A boolean value.

**GetNextDeftemplate**

PURPOSE: Allows access to the `ListOfDeftemplates`.

ARGUMENTS: A pointer to a deftemplate in the `ListOfDeftemplates`.

RETURNS: If passed a NULL pointer, returns the first deftemplate in the `ListOfDeftemplates`. Otherwise, returns the next deftemplate following the deftemplate passed as an argument.

**InitializeDeftemplates**

PURPOSE: Initializes the deftemplate construct. Adds `clear`, `save`, `bload`, `bsave`, and `constructs-to-c` functions for defglobals, and calls `DeftemplateCommands` to define functions associated with deftemplates.

**IsDeftemplateDeletable**

PURPOSE: Indicates whether a deftemplate can be deleted.

ARGUMENTS: A pointer to a deftemplate.

RETURNS: Boolean value. TRUE if the deftemplate can be deleted, otherwise FALSE.

**ListDeftemplates**

PURPOSE: Displays the `ListOfDeftemplates`.
**ListDeftemplatesCommand**

PURPOSE: Implements the `list-deftemplates` command. Uses the driver function `ListDeftemplates`.

**ModifyCommand**

PURPOSE: Implements the `modify` command. Calls the function `DuplicateModifyCommand` with a value of TRUE to execute the command.

**PPDeftemplate**

PURPOSE: Pretty prints a deftemplate.

ARGUMENTS: Name of deftemplate to be pretty printed and logical name of the output source.

**PPDeftemplateCommand**

PURPOSE: Implements the `ppdeftemplate` command. Uses the driver function `PPDeftemplate`.

**PrintTemplateFact**

PURPOSE: Prints a fact using the deftemplate format for displaying facts.

ARGUMENTS: A logical name to send the output and a pointer to a fact.

RETURNS: Boolean value. TRUE if the fact was successfully printed using the deftemplate format (i.e. a deftemplate was found that corresponded to the first field of the fact), otherwise FALSE.

**QFindDeftemplate**

PURPOSE: Finds a named deftemplate in the `ListOfDeftemplates`.

ARGUMENTS: The name of deftemplate to be found. This argument is specified as a pointer to a `SymbolTable` entry rather than a character string.

RETURNS: A pointer to the deftemplate if found, otherwise NULL.

**QSetListOfDeftemplates**

PURPOSE: Sets the `ListOfDeftemplates` to the specified value.

ARGUMENTS: A pointer to a linked list of deftemplates.
ReturnSlots

PURPOSE: Returns a linked list of slot data structures to the CLIPS memory manager.

ARGUMENTS: A linked list of slot data structures.

SDDCommand

PURPOSE: Implements the `set-dynamic-deftemplate-checking` command.

SetDynamicDeftemplateChecking

PURPOSE: Sets the current value of the `DynamicDeftemplateChecking` flag.

ARGUMENTS: A boolean value (the new value of the flag).

RETURNS: A boolean value (the old value of the flag).

SetListOfDeftemplates

PURPOSE: Sets the `ListOfDeftemplates` to the specified value. Normally used when initializing a run-time module or when blooding a binary file to install the `ListOfDeftemplates`. Adds each deftemplate in the new list to the `DeftemplateHashTable` by calling the function `AddHashDeftemplate`.

ARGUMENTS: A pointer to a linked list of deftemplates.

UndeftemplateCommand

PURPOSE: Implements the `undeftemplate` command.

INTERNAL FUNCTIONS

AddHashDeftemplate

PURPOSE: Adds a deftemplate to the `DeftempateHashTable`.

ARGUMENTS: A pointer to a deftemplate.

DeftemplateCommands

PURPOSE: Defines the commands `undeftemplate`, `ppdeftemplate`, `list-deftemplates`, `duplicate`, `modify`,
get-dynamic-deftemplate-checking, and set-dynamic-deftemplate-checking.

### Deftemplate Bload/Bsave Functions

**PURPOSE:** A set of functions used by the `bload` and `bsave` commands to process the deftemplate construct. These functions are made available to the `bload` and `bsave` commands by calling the function `AddBinaryItem`.

### Deftemplate Constructs-To-C Functions

**PURPOSE:** A set of functions used by the `constructs-to-c` command to process the deftemplate construct. These functions are made available to the `constructs-to-c` command by calling the function `AddCodeGeneratorItem`.

### DeinstallDeftemplate

**PURPOSE:** Decrements all occurrences in the `SymbolTable` of symbols found in an deftemplate, calls `DeinstallExpression` for all expressions used by the deftemplate, and removes the deftemplate from the `DeftempateHashTable`.

**ARGUMENTS:** A pointer to a deftemplate.

### DuplicateModifyCommand

**PURPOSE:** Implements the `duplicate` and `modify` commands. The fact being duplicated or modified is first copied to a new fact. Replacements to the fields of the new fact are then made. If a `modify` command is being performed, the original fact is retracted. Lastly, the new fact is asserted.

**ARGUMENTS:** Boolean value. If TRUE, the original fact used to duplicate the newly asserted fact is retracted (for the `modify` command). If FALSE, the original fact is not retracted (for the `duplicate` command). Other arguments to the command are retrieved using the argument access functions.

### InitializeDeftemplateHashTable

**PURPOSE:** Initializes the `DeftemplateHashTable`.

### InstallDeftemplate

**PURPOSE:** Increments all occurrences in the `SymbolTable` of symbols found in an deftemplate, calls `InstallExpression` for all
expressions used by the deftemplate, and adds the
deftemplate to the `DeftemplateHashTable`.

**ARGUMENTS:** A pointer to a deftemplate.

### RemoveHashDeftemplate

**PURPOSE:** Removes a deftemplate from the `DeftemplateHashTable`.

**ARGUMENTS:** A pointer to a deftemplate.

**RETURNS:** Boolean value. TRUE if the deftemplate was removed, otherwise FALSE.

### SaveDeftemplates

**PURPOSE:** Deftemplate construct save function. Pretty prints all
deftemplates to the given logical name.

**ARGUMENTS:** A logical name to send output.

### TemplateMultifieldSlotReplace

**PURPOSE:** Replaces the multifield value slot of a deftemplate fact. Called by `DuplicateModifyCommand` when replacing slot values.

**ARGUMENTS:** A pointer to the list of expressions to be evaluated and stored in the multifield slot, and pointer to the fact in which the new multifield value is to be stored, and a pointer to the deftemplate corresponding to the fact.

**RETURNS:** A pointer to a fact. If the previous size of the fact did not exactly match the exact amount of space needed for the new multifield value, then a new fact of the exact size is dynamically allocated and the old fact is discarded.
The Deftemplate Functions Module (deftmfun.c) provides parsing routines for the modify and duplicate commands and the template patterns used in the assert command. Template patterns found within these commands are referred to as RHS patterns (since these commands are typically used from the RHS of a rule). For a description of the modify and duplicate commands and using template patterns within the assert command, see the CLIPS Basic Programming Guide. The Deftemplate Function Module also provides the functionality for dynamic deftemplate checking. The deftemplate construct capability can be removed by using the appropriate compiler flag in the setup header file.

Template patterns are converted by CLIPS into regular positional patterns through a set of simple translation rules. By using field keywords, the fields of a template pattern can be specified in any order. The keywords fields, however, will be translated into a fixed positional order. The single-field values of a RHS template pattern retain the same order that they are given in their corresponding deftemplate. A multifield value in a RHS template pattern is always placed at the end of a pattern regardless of the position of the field definition in the deftemplate. For example, given the following deftemplate,

```lisp
(deftemplate example
  (multifield z)
  (field x (default 3))
  (field y))
```

The RHS pattern used in the following assert

```lisp
(assert (example (z c d e) (y b) (x a)))
```

would be translated to

```lisp
(assert (example a b c d e))
```

If a value is not specified in a template pattern, an appropriate value for the pattern will be used. A RHS template pattern used with an assert will use the default value for any unspecified fields. If a default value was not specified for a field, CLIPS provides a default value based on the allowed types for the field and the cardinality of the field (single-field or multifield). Unspecified fields in a RHS template pattern for the modify or duplicate command will be replaced with the current value of the field for the fact being modified. For example, the RHS pattern used in the following assert

```lisp
(assert (example (y 4)))
```

would be translated to

```lisp
(assert (example 3 4))
```

**GLOBAL VARIABLES**

None.
INTERNAL VARIABLES

None.

GLOBAL FUNCTIONS

**CheckTemplateFact**

**PURPOSE:** Performs dynamic deftemplate checking on individual fields of a deftemplate facts.

**ARGUMENTS:** The type of the field being checked, the value of the field being checked, a pointer to the slot data structure to which the field is being assigned, and a pointer to the fact in which the field is contained.

**RETURNS:** Boolean value. TRUE if a constraint error is detected, otherwise FALSE. Also sets the value of EvaluationError if a constraint error is detected.

**OTHER NOTES:** Calls the functions CheckSlotType, CheckSlotRange, and CheckSlotAllowedValues to perform the constraint checking.

**DuplicateParse**

**PURPOSE:** Coordinates all actions necessary to parse the duplicate command.

**ARGUMENTS:** Logical name from which input is read and a pointer to the expression function call.

**RETURNS:** Expression representing the duplicate command.

**OTHER NOTES:** The expression returned by this function may later be modified by the function UpdateModifyDuplicate. This function uses the ModAndDupParse function to actually parse the command.

**GetMultiSlotPosition**

**PURPOSE:** Given the relation name of a fact (the first field), determines if the fact is a deftemplate fact and if so which position in the fact corresponds to the beginning of the multifield slot.

**ARGUMENTS:** The relation name of a fact (a pointer to a symbol).

**RETURNS:** An integer value. -1 if the relation name is not associated with a deftemplate, 0 if the relation name is associated with a deftemplate that does not contain a multifield slot, and the
position of the multifield slot (a value greater than 0) for deftemplates containing a multifield slot.

**ModifyParse**

**PURPOSE:** Coordinates all actions necessary to parse the `modify` command.

**ARGUMENTS:** Logical name from which input is read and a pointer to the expression function call.

**RETURNS:** Expression representing the `modify` command.

**OTHER NOTES:** The expression returned by this function may later be modified by the function `UpdateModifyDuplicate`. This function uses the `ModAndDupParse` function to actually parse the command.

**MultiIntoSingleFieldSlotError**

**PURPOSE:** Determines if a multifield value is being placed into a single field slot of a deftemplate fact.

**ARGUMENTS:** The positional index of the slot being changed and the relation name of the fact (a pointer to a symbol).

**RETURNS:** Nothing, however, the `EvaluationError` is set to TRUE and an error message is printed if a multifield value is placed in a single field slot.

**ParseAssertTemplate**

**PURPOSE:** Coordinates all actions necessary to parse template patterns found in assert commands and deffacts constructs.

**ARGUMENTS:** Logical name from which input is read, a pointer to a token structure in which scanned tokens are placed, a pointer to integer flag in which a boolean value is to be stored indicating whether a multifield value was used in the assert, a pointer to integer flag in which a boolean value is to be stored indicating whether an error occurred while parsing, a pointer to the symbol containing the deftemplate name that was parsed from the first field of the fact, an integer value indicating the type of token that terminates the template (e.g. right parenthesis), and a boolean flag indicating if only constants are allowed within the template (to be used with functions such as `load-facts`).
RETURNS: An expression representing the list of values to be asserted. The error flag and multifield value flag passed as parameters may also be set.

**UpdateModifyDuplicate**

PURPOSE: Changes the **modify** and **duplicate** commands such that the integer positions of the slots are stored in the command rather than the slot name. This allows quicker replacement of slots. This replacement can only take place when the deftemplate is specified using a fact-address bound on the LHS of a rule.

ARGUMENTS: A pointer to a **modify** or **duplicate** command and the name of the command being updated.

RETURNS: Boolean value. TRUE if the command was successfully modified (or could not be modified because a fact-address was not used) and FALSE if an error occurred while modifying the command. The expression passed as an argument is also directly modified.

**INTERNAL FUNCTIONS**

**AssertSlotsMultiplyDefined**

PURPOSE: Determines if the same slot was specified more than once in a deftemplate assert pattern.

ARGUMENTS: A pointer to a list of slot assignments.

RETURNS: Boolean value. TRUE if a slot was specified more than once, otherwise FALSE.

**CheckRHSSlotTypes**

PURPOSE: Checks the validity of a change to a slot as the result of an **assert**, **modify**, or **duplicate** command. This checking is performed statically (i.e. when the command is being parsed).

ARGUMENTS: A pointer to the list of values to be stored in a slot, a pointer to a slot data structure, and the name of the command modifying the slot.

RETURNS: Boolean value. FALSE if the values do not satisfy slot restrictions and TRUE if they do satisfy slot restrictions.
**FieldCheckTemplate**

**PURPOSE:** Checks a slot from a template fact to make sure that it doesn’t violate any of the deftemplate’s constraints.

**ARGUMENTS:** The type of the slot’s value, a pointer to the slot’s value, a pointer to the slot data structure, and a pointer to the fact from which the slot’s value was extracted.

**RETURNS:** Boolean value. TRUE if one of the deftemplate’s constraints is violated.

**GetSlotAssertValues**

**PURPOSE:** Returns the assigned slot value for a specified slot. If no slot value has been assigned, then a default value is returned.

**ARGUMENTS:** A pointer to a slot data structure and a pointer to a list of slot assignments.

**RETURNS:** A pointer to an expression.

**ModAndDupParse**

**PURPOSE:** Handles parsing of the *modify* and *duplicate* commands.

**ARGUMENTS:** Logical name from which input is read, a pointer to the expression function call, and the name of the command being parsed.

**RETURNS:** A pointer to an expression.

**OTHER NOTES:** Calls the function *GetAssertArgument* to retrieve individual values to be changed in the slots.

**ModifySlotsMultiplyDefined**

**PURPOSE:** Determines if the same slot was specified more than once in a *modify* or *duplicate* command.

**ARGUMENTS:** A pointer to a list of slot assignments.

**RETURNS:** Boolean value. TRUE if a slot was specified more than once, otherwise FALSE.

**ParseAssertSlotValues**

**PURPOSE:** Parses the values to be asserted for a specified slot.
ARGUMENTS: Logical name from which input is read, a pointer to a token structure in which scanned tokens are placed, a pointer to a slot data structure, a pointer to integer flag in which a boolean value is to be stored indicating whether a multifield value was used in the assert, a pointer to integer flag in which a boolean value is to be stored indicating whether an error occurred while parsing, and a boolean flag indicating if only constants are allowed as slot values.

RETURNS: A pointer to the list of assert values for the slot. The error flag and multifield value flag passed as parameters may also be set.

OTHER NOTES: Used by the ParseAssertTemplate function. Calls the function GetAssertArgument to retrieve individual values to be stored in the slot.

---

ParseSlotLabel

PURPOSE: Parses the beginning of a slot definition. Checks for an opening left parenthesis and a valid slot name.

ARGUMENTS: Logical name from which input is read, a pointer to a token structure in which scanned tokens are placed, a pointer to a deftemplate data structure (for determining slot validity), a pointer to integer flag in which a boolean value is to be stored indicating whether an error occurred while parsing, and an integer value indicating the type of token that terminates the template (e.g. right parenthesis).

RETURNS: A pointer to the slot data structure referenced by the slot label. The error flag passed as a parameter may also be set.

OTHER NOTES: Used by the ParseAssertTemplate function.

---

ReorderAssertSlotValues

PURPOSE: Reorders a list of slot assignments for an assert command from a template ordering to a positional ordering. Unspecified slots are assigned default values.

ARGUMENTS: A pointer to a deftemplate data structure and a pointer to a list of slot assignments.

RETURNS: Returns a list of expressions which can directly be attached as the argument list to an assert command.
<table>
<thead>
<tr>
<th>ReturnSAPs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PURPOSE:</strong></td>
</tr>
<tr>
<td><strong>ARGUMENTS:</strong></td>
</tr>
</tbody>
</table>
The Deftemplate Parser Module (deftmpsr.c) contains the function necessary for parsing a deftemplate construct. The BNF for the deftemplate construct is shown in Appendix G of the CLIPS Basic Programming Guide.

GLOBAL VARIABLES
None.

INTERNAL VARIABLES

<table>
<thead>
<tr>
<th>DeftemplateError</th>
</tr>
</thead>
</table>
PURPOSE: Indicates whether an error was encountered during parsing of a deftemplate construct.

GLOBAL FUNCTIONS

<table>
<thead>
<tr>
<th>ParseDeftemplate</th>
</tr>
</thead>
</table>
PURPOSE: Coordinates all actions necessary for the construction of a deftemplate into the current environment. Called to parse a deftemplate construct.
ARGUMENTS: Logical name from which deftemplate input is read.
OTHER NOTES: Uses the functions GetConstructNameAndComment and SlotDeclarations to perform the parsing.

INTERNAL FUNCTIONS

<table>
<thead>
<tr>
<th>CheckSlotConflicts</th>
</tr>
</thead>
</table>
PURPOSE: Checks to determine if any of a slot's attributes conflict with each other (e.g. declaring a slot to be of type integer but declaring the default value to be a symbol).
ARGUMENTS: A pointer to the slot declaration structure.
RETURNS: Boolean value. FALSE indicates a conflict exists while TRUE indicates no conflict exists.

<table>
<thead>
<tr>
<th>DefinedSlots</th>
</tr>
</thead>
</table>
PURPOSE: Builds a new slot declaration and coordinates the parsing of the slot's attributes.
ARGUMENTS: Logical name from which input is read, the name of the slot, a boolean flag indicating if the slot is a multifield, and a pointer to a token structure in which scanned tokens are placed.

RETURNS: A pointer to the data structure which represents the slot declaration.

OTHER NOTES: Uses the ParseRangeAttribute, ParseTypeAttribute, ParseDefault, and ParseAllowedValuesAttribute functions to parse individual slot attributes.

### MultiplyDefinedSlots

**PURPOSE:** Determines if two slots in a deftemplate construct have been given identical names.

**ARGUMENTS:** A pointer to a list of slot declaration structures.

**RETURNS:** Boolean value. TRUE indicates that slots have been multiply defined, while FALSE indicates that slots have not been multiply defined.

### ParseAllowedValuesAttribute

**PURPOSE:** Parses the allowed-values, allowed-integers, allowed-floats, allowed-numbers, allowed-symbols, and allowed-strings slot attributes.

**ARGUMENTS:** Logical name from which input is read, a pointer to the slot declaration structure, and the name of the slot attribute being parsed.

**RETURNS:** Boolean value. FALSE indicates an error occurred while parsing the type attribute, TRUE indicates no error occurred. The range fields of the slot declaration structure are also modified based upon the range declaration.

### ParseDefault

**PURPOSE:** Parses the default slot attribute.

**ARGUMENTS:** Logical name from which input is read, a boolean flag indicating if the slot is a multifield, and a pointer to a token structure in which scanned tokens are placed.

**RETURNS:** A pointer to an expression containing the default slot value(s).
**ParseRangeAttribute**

**PURPOSE:** Parses the range slot attribute.

**ARGUMENTS:** Logical name from which input is read and a pointer to the slot declaration structure.

**RETURNS:** Boolean value. FALSE indicates an error occurred while parsing the type attribute, TRUE indicates no error occurred. The range fields of the slot declaration structure are also modified based upon the range declaration.

**ParseSlot**

**PURPOSE:** Coordinates the parsing of an individual slot declaration.

**ARGUMENTS:** Logical name from which input is read and a pointer to a token structure in which scanned tokens are placed.

**RETURNS:** A pointer to the data structure which represents the slot declaration.

**OTHER NOTES:** Uses the function `DefinedSlots` to coordinate parsing of individual slot attributes.

**ParseTypeAttribute**

**PURPOSE:** Parses the type slot attribute.

**ARGUMENTS:** Logical name from which input is read and a pointer to the slot declaration structure.

**RETURNS:** Boolean value. FALSE indicates an error occurred while parsing the type attribute, TRUE indicates no error occurred. The allowed type fields of the slot declaration structure are also modified based upon the type declaration.

**SlotDeclarations**

**PURPOSE:** Coordinates the parsing of all of the slot declarations (i.e. field and multifield specifications) for a deftemplate construct.

**ARGUMENTS:** Logical name from which input is read and a pointer to a token structure in which scanned tokens are placed.

**RETURNS:** A linked list of data structures which represent the slot declarations.
OTHER NOTES: Uses the function ParseSlot to parse individual slot declarations.
The Deftemplate LHS Module (deftmlhs.c) provides routines for parsing deftemplate pattern found in the LHS of a rule. For a description of using template patterns on the LHS of a rule, see the CLIPS Basic Programming Guide.

Template patterns are converted by CLIPS into regular positional patterns through a set of simple translation rules. By using field keywords, the fields of a template pattern can be specified in any order. The keywords fields, however, will be translated into a fixed positional order. The single-field values of a LHS template pattern retain the same order that they are given in their corresponding deftemplate. A multifield value in a LHS template pattern is always placed at the end of a pattern regardless of the position of the field definition in the deftemplate. For example, given the following deftemplate,

```lisp
(deftemplate example
  (multifield z)
  (field x (default 3))
  (field y))
```

the LHS pattern shown following

```lisp
(example (z $?z) (y ?y) (x ?x))
```

would be translated to

```lisp
(example ?x ?y $?z)
```

If a value is not specified in a template pattern, an appropriate value for the pattern will be used. If a LHS template pattern has an unspecified value, that value will be replaced with ? for single-field slots and $? for multifield slots. For example, the LHS template pattern shown following

```lisp
(example (y 3))
```

would be translated to

```lisp
(example ? 3 $)
```

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

None.
GLOBAL FUNCTIONS

**DeftemplateLHSParse**

**PURPOSE:** Parses a deftemplate pattern found on the LHS of a rule.

**ARGUMENTS:** Logical name from which input is read and the name of the deftemplate corresponding to the first field of the pattern being parsed.

**RETURNS:** A pointer to a linked structure containing the intermediate LHS representation of the pattern. If an error has occurred during parsing, a NULL pointer is returned.

**OTHER NOTES:** The intermediate LHS representation is built using the functions `GetLHSSlots`, `MultiplyDefinedLHSSlots`, and `ReorderLHSSlotValues`.

INTERNAL FUNCTIONS

**CheckLHSSlotTypes**

**PURPOSE:** Determines if a slot constraint satisfies slot restrictions (e.g. type, range, and value).

**ARGUMENTS:** A pointer to the slot constraints for a slot and a pointer to a corresponding slot data structure.

**RETURNS:** Boolean value. TRUE, if the slot restrictions are satisfied (i.e. have not been violated), otherwise FALSE.

**GetLHSSlots**

**PURPOSE:** Parses the list of slot constraints associated with a LHS deftemplate pattern.

**ARGUMENTS:** Logical name from which input is read, a pointer to a token structure in which scanned tokens are placed, a pointer to the deftemplate data structure for the pattern being parsed, and a pointer to an integer flag in which an error status is stored.

**RETURNS:** Returns a linked list of the slot constraints for the pattern (which could be a NULL pointer if there are no slot constraints). If an error occurs, the integer flag passed as an argument is set to TRUE.

**OTHER NOTES:** Builds the list of slot constraints using the functions `GetSingleLHSSlot` and `CheckLHSSlotTypes`.

230  Deftemplate LHS Module
**GetSingleLHSSlots**

**PURPOSE:** Parses a single slot constraint associated with a LHS deftemplate pattern.

**ARGUMENTS:** Logical name from which input is read, a pointer to a token structure in which scanned tokens are placed, a pointer to the slot data structure for the slot being parsed, and a pointer to an integer flag in which an error status is stored.

**RETURNS:** Returns the slot constraints for the slot. If an error occurs, the integer flag passed as an argument is set to TRUE and a NULL pointer is returned.

**OTHER NOTES:** Primarily uses the function `RestrictionParse` to parse the slot constraints.

**MultiplyDefinedLHSSlots**

**PURPOSE:** Determines if two slot constraints in a deftemplate LHS pattern have been given for the same slot name.

**ARGUMENTS:** A pointer to a list of slot constraints.

**RETURNS:** Boolean value. TRUE indicates that slots have been multiply defined, while FALSE indicates that slots have not been multiply defined.

**ReorderLHSSlotValues**

**PURPOSE:** Reorders a list of slot constraints for a LHS deftemplate pattern from a template ordering to a positional ordering. Unspecified slots match against wildcards (for single field slots) and multifield wildcards (for multifield slots).

**ARGUMENTS:** A pointer to the list of slots for a deftemplate, a pointer to the list of specified slot constraints for the pattern, and a pointer to the current intermediate LHS representation of the deftemplate pattern (which contains only the deftemplate name).

**RETURNS:** A pointer to the LHS representation of the deftemplate pattern containing the slot constraints in the appropriate order with unspecified slots replaced with wildcards.

**ReturnSLPs**

**PURPOSE:** Returns intermediate data structures used for parsing an deftemplate LHS pattern to the CLIPS memory manager.
ARGUMENTS: A pointer to a list of slot constraints.
Binary Save Module

The Binary Save Module (bsave.c) provides the functionality for the bsave command. To illustrate how a binary save is performed, the following code will be used as example:

```
(deffacts start-info
  (point 3.7 5.3))

(deffunction compute ()
  (+ (* 2 3) (* 9.1 2.3)))
```

For future reference, the expressions associated with the deffacts start-info and the deffunction compute are show below. Note that there are some differences between the representation shown below and the actual representation in CLIPS. For example, the body of a deffunction is enclosed within a `progn` function, but this is not shown in diagram below.

In the proceeding descriptions, it will be assumed that integers and pointers occupy four bytes of memory storage and that double precision floating pointer numbers occupy four bytes of memory storage. Characters always occupy one byte of memory storage regardless of the C implementation.

The first step in creating a binary image file is to write a binary header to the file so that the bsave command can verify that the file is CLIPS binary file and determine...
which version of CLIPS created the file. The binary header is written in two distinct parts. The first part indicates that the file is a CLIPS binary file and contains control characters to help prevent a text file as being mistaken for a binary file. The second part contains the version number of CLIPS that was used to create the binary file. An example binary header is shown following:

```
"\1\2\3\4CLIPS"  "V5.00"
```

In the second step, all functions, symbols, integers, and floats within CLIPS have a status flag set to FALSE to indicate that each of these items is not needed by the binary image. Then, each construct that has been registered using the `AddBinaryItem` function has its specified function called which is used to mark which functions, symbols, integers, floats are need to save the binary image for that construct. Finally, after all constructs have had the opportunity to mark needed items, each needed symbol, integer, and float is assigned a unique integer id that will be used as a reference in the binary image. In the example shown previously, the functions `assert`, `+`, and `*` are needed, the symbols `start-info`, `compute`, and `point` are needed, the floats 3.7, 5.3, 9.1, and 2.3 are needed, and the integers 2 and 3 are needed.

In step three, each of the needed functions is written out to the binary image. The number of functions needed is written, followed by the amount of space for all of the function names, followed by each of the function names as a NULL terminated string. Functions are given indexes at this point. Note that these are not written in any specific order with relation to which binary item needs them. A function is only written once (regardless of the number of binary items which use it—it’s either needed or its not). For the example shown previously, the following output would be written to the binary file.

```
3 4 bytes: Number of Functions
11 4 bytes: Total Size of Function Names
"+" 2 bytes: Space for Function #0
"*" 2 bytes: Space for Function #1
"assert" 7 bytes: Space for Function #2
```

In step four, each of the needed symbols is written out to the binary image. The number of symbols needed is written, followed by the amount of space for all of the symbol names, followed by each of the symbol names as a NULL terminated string. For the example shown previously, the following output would be written to the binary file.
In step five, each of the needed floats is written out to the binary image. The number of floats needed is written followed by each of the floats needed. For the example shown previously, the following output would be written to the binary file.

<table>
<thead>
<tr>
<th>3</th>
<th>4 bytes: Number of Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>4 bytes: Total Size of Symbols</td>
</tr>
<tr>
<td>&quot;start-info&quot;</td>
<td>12 bytes: Space for Symbol #0</td>
</tr>
<tr>
<td>&quot;compute&quot;</td>
<td>8 bytes: Space for Symbol #1</td>
</tr>
<tr>
<td>&quot;point&quot;</td>
<td>6 bytes: Space for Symbol #2</td>
</tr>
</tbody>
</table>

In step six, each of the needed integers is written out to the binary image. The number of integers needed is written followed by each of the integers needed. For the example shown previously, the following output would be written to the binary file.

<table>
<thead>
<tr>
<th>4</th>
<th>4 bytes: Number of Floats</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>8 bytes: Space for Float #0</td>
</tr>
<tr>
<td>2.3</td>
<td>8 bytes: Space for Float #1</td>
</tr>
<tr>
<td>3.7</td>
<td>8 bytes: Space for Float #3</td>
</tr>
<tr>
<td>5.3</td>
<td>8 bytes: Space for Float #4</td>
</tr>
</tbody>
</table>

In step seven, each of the expressions needed by the constructs is written out to the binary image. First, the total number of expression structures is written out to the binary image and then each registered construct is called to dump its expressions to the file. Each expression is written to the binary image using the following format.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>INDEX VALUE</th>
<th>ARGUMENT LIST</th>
<th>NEXT ARGUMENT</th>
</tr>
</thead>
</table>

The type field corresponds directly to its value in the CLIPS expression data structure. The index value, argument list, and next argument fields are pointers in the CLIPS expression data structure, but are converted to integers indexes when saved to the binary image. For example, the index value for the symbol point would be 2 since it is the third item saved in the symbol section of the binary image (note that the nth item in a C array is referenced by n-1). Pointers other expressions or data structures are converted to integer indexes as well. A NULL pointer is converted to the value -1. For the example shown previously, the following output would be written to the binary file for saving the expressions.
In step eight, each registered construct is written to the binary image. First, the name of the construct is written to the binary image (up to a specified number of characters). The bsave function for the construct is then called. The bsave function for the construct is first responsible for writing the amount space required by it (in case the construct must be skipped over when bloaded). It then can write out any data structures that it needs.

If the deffacts construct was defined using the following C data structure,

```c
struct deffacts
{
    struct symbol *name;
    char *ppForm;
    struct expression *assertItems;
    struct deffacts *next;
};
```

then the following format could be used in writing the deffacts constructs to the binary image.

<table>
<thead>
<tr>
<th>NAME</th>
<th>PRETTY PRINT</th>
<th>DEFFACTS EXPRESSION</th>
<th>NEXT DEFFACTS</th>
</tr>
</thead>
</table>

The `name` field would be the integer index of a symbol, the `ppForm` field could be given an index of -1 (since pretty print forms are not loaded with binary images), the `assertItems` field would be given the appropriate index for the expression saved previously, and the `next` field would contain the integer index to the next deffacts in the binary image (-1 for the last deffacts). For the example shown previously, the following output would be written to the binary file for saving the deffacts.
Similarly, if the deffunction construct was defined using the following C data structure,

```c
struct deffunction
{
  struct symbol *name;
  char *ppForm;
  struct expression *code;
  int numberOfParameters;
  struct deffunction *next;
};
```

then the following format could be used in writing the deffunction constructs to the binary image.

<table>
<thead>
<tr>
<th>NAME</th>
<th>PRETTY PRINT</th>
<th>DEFFUNCTION BODY</th>
<th>NUMBER OF ARGUMENTS</th>
<th>NEXT DEFFUNCTION</th>
</tr>
</thead>
</table>

The `name` field would be the integer index of a symbol, the `ppForm` field could be given an index of -1 (since pretty print forms are not loaded with binary images), the `code` field would be given the appropriate index for the expression saved previously, the `numberOfParameters` field would be given its actual integer value, and the `next` field would contain the integer index to the next deffunction in the binary image (-1 for the last deffunction). For the example shown previously, the following output would be written to the binary file for saving the deffunctions.

<table>
<thead>
<tr>
<th>&quot;deffunction&quot;</th>
<th>20 bytes: Deffunction Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>4 bytes: Total Size of Deffunctions</td>
</tr>
<tr>
<td>1</td>
<td>4 bytes: Number of Deffunctions</td>
</tr>
<tr>
<td>1 -1 4 0 -1</td>
<td>20 bytes: Deffacts #0</td>
</tr>
</tbody>
</table>

The final step in saving a binary image is to write a binary footer to identify the end of the binary image. This binary footer is identical to the first part of the binary header. An example binary footer is shown following:

"\1\2\3\4CLIPS"
GLOBAL VARIABLES

<table>
<thead>
<tr>
<th>ExpressionCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: An integer value representing the number of expression data structures which have been encountered or processed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ListOfBinaryItems</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Contains a list of data structures used to call functions for various constructs which generate output or read input for the bsave/ reload commands.</td>
</tr>
</tbody>
</table>

INTERNAL VARIABLES

None.

GLOBAL FUNCTIONS

<table>
<thead>
<tr>
<th>AddBinaryItem</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Adds an item to the ListOfBinaryItems. Every construct that generates output for bsave must use this function to install functions that will be called whenever a bsave command is executed.</td>
</tr>
<tr>
<td>ARGUMENTS: A name to be associated with the binary item, the priority of the item, a pointer to a function which is called to mark items needed by the binary item (symbols, floats, integers, and functions), a pointer to function which will write all expressions needed by a binary item to the binary output file, a pointer to a function which writes other data required by a binary item to the binary output file, a pointer to a function which is call to load input from a binary file when a bload is performed, a pointer to a function which initializes the data for a binary item after a bload has been performed (e.g. converting indices to actual addresses), and a pointer to a function which is called when a clear command is executed while a binary image is loaded.</td>
</tr>
<tr>
<td>RETURNS: Boolean value. TRUE if the binary item was successfully added, otherwise FALSE.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bsave</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Main driver routine for coordinating binary output to a file for the bsave function.</td>
</tr>
</tbody>
</table>
ARGUMENTS: The name of the file in which to store the binary representation of the constructs currently loaded in the CLIPS environment.

RETURNS: Boolean value. TRUE, if the binary file was successfully created, otherwise FALSE.

BsaveCommand

PURPOSE: Converts CLIPS constructs currently loaded into the CLIPS environment into a binary representation stores them in a file. This function is the driver routine for the CLIPS function bsave. The function Bsave is called by this function to perform the code generation.

ARGUMENTS: No actual arguments. The CLIPS arguments passing routines are used to extract arguments when this function is called from the CLIPS environment.

BsaveExpression

PURPOSE: Writes the binary representation of an expression to a file.

ARGUMENTS: A pointer to a file and a pointer to an expression.

OTHER NOTES: Uses and increments the value of the global variable ExpressionCount to convert the pointers (to other expressions) found in the expression to integer index values. Symbols, floats, integers, functions, and pointers to other constructs (such as generic functions) already have index values computed for them. The index values for the expression are converted back to addresses when a bload is performed.

GenWrite

PURPOSE: Provides a generic capability for writing binary output to a file.

ARGUMENTS: A pointer to the data to be written, the length of the data, and a pointer to a file.

MarkNeededItems

PURPOSE: Given an expression, marks the symbols, floats, integers, and functions contained within the expression as being needed by this binary image.

ARGUMENTS: A pointer to an expression.
INTERNAL FUNCTIONS

BsaveAllExpressions

PURPOSE: Called by function Bsave to write the binary representation of all expressions used by this binary image. Calls the expression function for each entry in the ListOfBinaryItems to allow the binary entry to write out needed expressions to the binary file.

ARGUMENTS: A pointer to a file.

OTHER NOTES: Sets the value of ExpressionCount to zero before calling any of the expression functions for the binary items. This value is then incremented as each binary item calls BsaveExpression to save its expressions.

FindNeededFunctionsAndAtoms

PURPOSE: Calls the find function for each entry in the ListOfBinaryItems to allow the binary entry to mark the symbols, floats, integers, and functions which should be saved as part of the binary image since they are needed by the entry. Conversely unneeded symbols, floats, integers, and functions will not be marked and thus will not be saved as part of the binary image. This is important since it allows binary images to be saved from versions of CLIPS customized with user defined functions and then loaded into non-customized versions as long as the user defined functions were not referenced by the saved binary image.

FunctionBinarySize

PURPOSE: Computes the total number of bytes in string space for all the required function names needed by the binary image.

RETURNS: A long integer.

MarkBuckets

PURPOSE: Replaces the bucket slot of each entry in the SymbolTable, IntegerTable, and FloatTable with an integer index that will be used to refer to that value. For example, the seventh symbol in the SymbolTable has the value 7 stored in its bucket slot (which would normally indicate the location in the SymbolTable that the symbol is stored).
**MarkNeededFlags**

**PURPOSE:** Marks every symbol, float, integer, and function as being unneeded by this binary image. This function is called before the binary items in the **ListOfBinaryItems** are allowed to mark symbols, floats, integers, and functions as being needed.

**UnmarkBuckets**

**PURPOSE:** Restores the bucket slot of each entry in the **SymbolTable**, **IntegerTable**, and **FloatTable** with its appropriate value. This function is called to reverse the changes made by **MarkBuckets**.

**WriteBinaryFooter**

**PURPOSE:** Writes the global variable **BinaryPrefixID** to the specified file to indicate that the end of a binary file has been reached.

**ARGUMENTS:** A pointer to a file.

**WriteBinaryHeader**

**PURPOSE:** Writes the global variables **BinaryPrefixID** and **BinaryVersionID** to the specified file to indicate that the file is a binary file. The **BinaryPrefixID** is used to indicate that the file is a binary file and the **BinaryVersionID** is used to indicate which version of CLIPS created the file.

**ARGUMENTS:** A pointer to a file.

**WriteNeededIntegers**

**PURPOSE:** Called by function **Bsave** to write the binary representation of all the integers required by this binary image.

**ARGUMENTS:** A pointer to a file.

**OTHER NOTES:** The number of integers required by this image is written to the binary file followed by the integers written in their binary representation (not their ASCII string representation).

**WriteNeededFloats**

**PURPOSE:** Called by function **Bsave** to write the binary representation of all the floats required by this binary image.

**ARGUMENTS:** A pointer to a file.
OTHER NOTES: The number of floats required by this image is written to the binary file followed by the floats written in their binary representation (not their ASCII string representation).

---

### WriteNeededFunctions

**PURPOSE:** Called by function `Bsave` to write the binary representation of all the functions required by this binary image. This function also assigns each required function an integer index which is used in place of the function’s address to refer to the function in the binary image.

**ARGUMENTS:** A pointer to a file.

**OTHER NOTES:** The number of functions required by this image is written to the binary file, followed by the total amount of space in bytes of all the required function names, followed by the names of the functions written as C strings.

---

### WriteNeededSymbols

**PURPOSE:** Called by function `Bsave` to write the binary representation of all the symbols required by this binary image.

**ARGUMENTS:** A pointer to a file.

**OTHER NOTES:** The number of symbols required by this image is written to the binary file, followed by the total amount of space in bytes of all the required symbols, followed by the symbols written as C strings.
The Binary Load Module (bload.c) provides the functionality for the bload command. Binary Images which are loaded must first have been saved as a binary image using the Binary Save Module (bsave.c). The following steps describe how a binary image is loaded. A knowledge of how the binary save works is assumed.

The first step in loading a binary image is to load the binary header from the file to determine if indeed the file is a binary image and whether it was created by the same version of CLIPS that is loading it. Next the CLIPS environment is cleared. The needed functions names are then loaded from the binary image. If any of these functions are unavailable, then the binary load is aborted. It is possible to save a binary image from a customized version of CLIPS which has additional functions not available in the standard version of CLIPS and then load the binary image in a standard or differently customized version of CLIPS as long as no customized functions are used. After the needed function names have been loaded, the need symbols, floats, and integers are loaded from the binary image, followed by all of the needed expressions.

Once these basic items have been loaded from the binary image, construct information is then loaded. A construct name is loaded from the binary image and then the appropriate function for the registered construct is called to load the construct’s binary image. If the construct is not registered, then the construct information in the binary image is skipped. This makes it possible to load part of a binary image into a customized version of CLIPS if all the needed constructs are not available. After a construct’s binary image is read, the process is repeated to load another construct. This process continues until the binary footer is encountered.

Finally, integer index references to symbols, floats, integers, and expressions are replaced with actual pointer values to the specified data structures. Each registered construct is also allowed to replace integer index references to data structures that it references.

GLOBAL VARIABLES

<table>
<thead>
<tr>
<th>BinaryPrefixID</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: The character string that is placed at the beginning and end of a binary file.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BinaryVersionID</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: The character string BinaryVersionID placed after the BinaryPrefixID at the beginning of a binary file to indicate which version of CLIPS created the file.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ExpressionArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: The array containing the expressions used by the current binary image.</td>
</tr>
</tbody>
</table>
**FloatArray**

**PURPOSE:** The array containing pointers to the floats required by the current binary image.

**FunctionArray**

**PURPOSE:** The array containing pointers to the functions required by the current binary image.

**IntegerArray**

**PURPOSE:** The array containing pointers to the integers required by the current binary image.

**SymbolArray**

**PURPOSE:** The array containing pointers to the symbols required by the current binary image.

**INTERNAL VARIABLES**

**AbortBloadFunctions**

**PURPOSE:** Contains a list of functions to be called whenever a `bload` command is aborted due to an error.

**AfterBloadFunctions**

**PURPOSE:** Contains a list of functions to be after a binary image has been loaded and refreshed during a `bload` command. These functions are used for initialization unrelated to loading the binary image (such as incremental reset for rules).

**BeforeBloadFunctions**

**PURPOSE:** Contains a list of functions to be called before a binary image is loaded during a `bload` command. These functions are called after the environment has been cleared, but before the binary image is loaded.

**BinaryFileHandle**

**PURPOSE:** Reference value for the binary file currently being loaded on the IBM PC.
**BinaryRefNum**

**PURPOSE:** Reference value for the binary file currently being loaded on the Macintosh.

**BinaryFP**

**PURPOSE:** Reference value for the binary file currently being loaded on machines other than the Macintosh or IBM PC.

**BloadActive**

**PURPOSE:** Boolean value indicating whether a binary image is currently loaded into the CLIPS environment.

**BloadReadyFunctions**

**PURPOSE:** Contains a list of functions to be called after the **BeforeBloadFunctions** are called to determine if the **bload** should continue or be aborted.

**ClearBloadReadyFunctions**

**PURPOSE:** Contains a list of functions to be called before a **clear** command of a binary image to determine if the clear should even be attempted.

**NumberOfExpressions**

**PURPOSE:** Integer value indicating the number of expression data structures contained in the **ExpressionArray**.

**GLOBAL FUNCTIONS**

**AddAbortBloadFunction**

**PURPOSE:** Adds a function to the list of **AbortBloadFunctions**.

**ARGUMENTS:** A name to be associated with the abort function, the priority of the function, and a pointer to a function to be called whenever a **bload** is aborted.

**AddAfterBloadFunction**

**PURPOSE:** Adds a function to the list of **AfterBloadFunctions**.
ARGUMENTS: A name to be associated with the abort function, the priority of the function, and a pointer to a function after a bload is performed.

AddBeforeBloadFunction

PURPOSE: Adds a function to the list of BeforeBloadFunctions.

ARGUMENTS: A name to be associated with the abort function, the priority of the function, and a pointer to a function before a bload is performed.

AddBloadReadyFunction

PURPOSE: Adds a function to the list of BloadReadyFunctions.

ARGUMENTS: A name to be associated with the abort function, the priority of the function, and a pointer to a function to call to determine if a bload command should be performed.

AddClearBloadReadyFunction

PURPOSE: Adds a function to the list of ClearBloadReadyFunctions.

ARGUMENTS: A name to be associated with the abort function, the priority of the function, and a pointer to a function to call to determine if the current binary image can be cleared.

Bload

PURPOSE: Main driver routine for coordinating the loading of a binary file for the bload function.

ARGUMENTS: The name of the CLIPS binary file to be loaded.

RETURNS: Boolean value. TRUE, if the binary file was successfully created, otherwise FALSE.

BloadCommand

PURPOSE: Loads a CLIPS binary file into the CLIPS environment. This function is the driver routine for the CLIPS function bload. The function Bload is called by this function to load the binary image.

ARGUMENTS: No actual arguments. The CLIPS arguments passing routines are used to extract arguments when this function is called from the CLIPS environment.
**Bload**

**PURPOSE:** Indicates whether a binary image is currently loaded in the CLIPS environment.

**OTHER NOTES:** Boolean value. Returns the value of the **BloadActive** variable.

**GenRead**

**PURPOSE:** Provides a generic capability for reading binary input from a file.

**ARGUMENTS:** A pointer to the storage area in which the data is to be read and an integer indicating the amount of data to read. The data is read from the file indicated by one of the variables **BinaryRefNum**, **BinaryFileHandle** or **BinaryFP** depending upon the machine on which CLIPS is running.

**INTERNAL FUNCTIONS**

**AbortBload**

**PURPOSE:** Handles error recovery if an error occurs while performing a **bload** command. Calls the abort function for each binary item.

**AddBloadFunctionToList**

**PURPOSE:** Adds a function to a list of functions. Called by routines such as **AddBloadReadyFunction**.

**ARGUMENTS:** A name to be associated with the abort function, the priority of the function, a pointer to a function, and a pointer to the head of the list on which the function is to be placed.

**RETURNS:** A pointer to the new head of the list of functions (if the function was added at the beginning of the list), otherwise the old head of the list is returned. In either case, the function is added to the list.

**BloadExpressions**

**PURPOSE:** Called by function **Bload** to load all of the expressions used by the binary image into an array.
RETURNS: A pointer to an array containing all the expression data structures used by this binary image. The global variable NumberOfExpressions is set by this function.

### ClearLoad

**PURPOSE:** Clears a binary image from the CLIPS environment.

**RETURNS:** Boolean value. TRUE if the binary image was successfully cleared, otherwise FALSE.

### FastFindFunction

**PURPOSE:** Searches for the data structure associated with a specified function name.

**ARGUMENTS:** A pointer to the name of the function and a pointer to a function data structure in the ListOfFunctions.

**RETURNS:** A pointer to the function data structure of the specified function if the function was found, otherwise NULL.

**OTHER NOTES:** This function is faster than FindFunction because the functions required by the binary image should have been written out in the same order that they appear in the the ListOfFunctions. Since the starting location of the search can be specified, the last value returned by FastFindFunction can be used as starting location for the next search.

### GenClose

**PURPOSE:** Provides a generic capability for closing a binary file.

**ARGUMENTS:** None. Closes the file specified by one of the global variables BinaryRefNum, BinaryFileHandle or BinaryFP depending upon the machine on which CLIPS is running.

### GenOpen

**PURPOSE:** Provides a generic capability for opening a binary file.

**ARGUMENTS:** The name of the file to be opened.

**RETURNS:** Boolean value. TRUE if the file was successfully opened, otherwise, FALSE. Also sets the machine specific global variables storing the pointers or values which refer to the file just opened. For the Macintosh, the variable BinaryRefNum is set. For the IBM PC, the variable
BinaryFileHandle is set. For all other machines, the variable BinaryFP is set.

### ReadNeededFloats

**PURPOSE:** Called by function Bload to generate an array containing the pointers to floats required by the binary image being loaded. The required floats are first loaded from the CLIPS binary file, then the function AddDouble is used to add the float to the FloatTable. A pointer to the float’s data structure is then stored in an array.

**ARGUMENTS:** A pointer to an integer in which the number of floats required by this binary image is stored.

**RETURNS:** A pointer to an array containing the pointers to floats required by this binary image. The number of floats argument is also assigned a value by this function.

### ReadNeededFunctions

**PURPOSE:** Called by function Bload to generate an array containing the pointers to functions required by the binary image being loaded. The names of the required functions are first loaded from the CLIPS binary file, then the function FastFindFunction is used to locate the data structure corresponding to each function name. A pointer to the function’s data structure is then stored in an array.

**ARGUMENTS:** A pointer to an integer in which the number of functions required by this binary image is stored and a pointer to an integer flag in which an error value is stored.

**RETURNS:** A pointer to an array containing the pointers to functions required by this binary image. Both arguments to this function are also assigned values. The error flag will be set to TRUE if any of the functions required by this binary image could not be found.

### ReadNeededIntegers

**PURPOSE:** Called by function Bload to generate an array containing the pointers to integers required by the binary image being loaded. The required integers are first loaded from the CLIPS binary file, then the function AddLong is used to add the integer to the IntegerTable. A pointer to the integer’s data structure is then stored in an array.
ARGUMENTS: A pointer to an integer in which the number of integers required by this binary image is stored.

RETURNS: A pointer to an array containing the pointers to integers required by this binary image. The number of integers argument is also assigned a value by this function.

**ReadNeededSymbols**

PURPOSE: Called by function **Bload** to generate an array containing the pointers to symbols required by the binary image being loaded. The required symbols are first loaded from the CLIPS binary file, then the function **AddSymbol** is used to add the symbol to the **SymbolTable**. A pointer to the symbol’s data structure is then stored in an array.

ARGUMENTS: A pointer to an integer in which the number of symbols required by this binary image is stored.

RETURNS: A pointer to an array containing the pointers to symbols required by this binary image. The number of symbols argument is also assigned a value by this function.

**RefreshExpressions**

PURPOSE: Converts all of the integer index values found in expressions in the **ExpressionArray** to actual addresses.
The Construct Compiler Module (conscomp.c) provides the functionality for the **constructs-to-c** command by generating a set of C source code files representing the constructs currently loaded in the CLIPS environment. The files can then be compiled and linked with a runtime version of CLIPS specifically compiled for use with files generated by the construct compiler. The runtime version of CLIPS is smaller than standard version of CLIPS since a significant amount of code used for parsing constructs is removed.

The construct compiler works in a manner similar to a binary save and load. However, instead of dumping CLIPS constructs to a file as binary data, the construct compiler dumps the C code representation of the constructs to a C source file which is converted to binary data by a compiler. Since the resulting object file can be linked directly with CLIPS, there is no need to load the constructs. In addition, saving a binary image requires that pointer references be converted to integer indexes which must be converted back to pointer references when the binary image is loaded. When generating C code using the construct compiler, pointer references can be directly expressed as pointer references which can be automatically resolved by the compiler and linker.

As an example of how the construct compiler works, assume that the following deffacts has been entered into the CLIPS knowledge base.

```clips
(deffacts start-info
  (point 3.7 5.3))
```

In addition, the following **initial-facts** deffacts is automatically entered into the CLIPS knowledge base.

```clips
(deffacts initial-fact
  (initial-fact))
```

Now, assume that the user entered the following command.

```clips
CLIPS> (constructs-to-c xmp 3)
CLIPS>
```

The first step taken by the construct compiler is to generate a header file which will be used by the C source files which will be generated. Since the symbol *xmp* was given a the prefix symbol, the header file “xmp.h” will be generated. Initially, all extern definitions for user and system defined functions are written to this header file. Later, as needed, other extern definitions for data structures will be written to this file so that other files can access the data structures.

Next, all symbols, floats, integers, and user-defined and system function definitions (those defined using **DefineFunction**) are written out to files. When creating file names, a naming convention is used by construct compiler. The first number appended to the file name prefix indicates the general contents of the file and the second number appended indicates the nth file of that type. For example, if three files were required to save all symbols and the number used to indicate symbol files was 2, then the files would be name “xmp2_1.c,” “xmp2_2.c,” and “xmp2_3.c.”
Once all basic data types have been written to files, each construct that was registered with the construct compiler using the `AddCodeGenItem` function is called so that it can generate its own C files. Each construct is only responsible for generating the code for the data structures that it has defined. References to symbols, floats, and integers are resolved by calling the functions `PrintSymbolReference`, `PrintFloatReference`, and `PrintIntegerReference` which will print the appropriate reference to the data item in the C file being generated by the construct. Each construct must also call the `ExpressionToCode` function for any expressions that it needs saved. This function will generate the appropriate code for the expression and print a reference to the expression in the C file being generated by the construct. Finally, although constructs are not required to, they should attempt to honor the limit on the maximum number of array elements which should be placed in a file.

Once all constructs have generated code, the initialization function is written and each construct is given the opportunity to add code to this function. For the example above, the initialization function would be named `InitImage_3`.

Returning to the deffacts example shown previously, if the deffacts data structure were defined as follows

```c
struct deffacts
{
    struct symbol *name;
    char *ppForm;
    struct expression *assertItems;
    struct deffacts *next;
};
```

then the following source code might be generated to represent all deffacts constructs.

```c
#include "xmp.h"

struct deffacts dft3_1[] = {
    {&shn3_1[166],NULL,&exp3_1[0],&dft3_1[1]},
    {&shn3_1[287],NULL,&exp3_1[2],NULL}};
```

The first line of code includes the header file “xmp.h”. This file is needed in order to access the data structures representing symbols and expressions. The array name `dft3_1` indicates that the image ID for the files generated by the construct compiler is 3 and that this is the first file containing deffacts structures. The reference `&shn3_1[166]` would be a pointer to the symbol data structure for the `initial-facts` symbol. Similarly, the reference `&shn3_1[287]` would be a pointer to the symbol data structure for the `start-info` symbol. The references `&exp3_1[0]` and `&exp3_1[2]` point to the expressions which assert the facts specified by the deffacts. Finally, the first `dft3_1` array element contains the reference `&dft3_1[1]` which is simply a pointer to the next deffacts in the list.

**GLOBAL VARIABLES**

None.
INTERNAL VARIABLES

**ExpressionCount**
PURPOSE: An integer value representing the number of expression data structures which have been written to the current *ExpressionFP* file.

**ExpressionFP**
PURPOSE: A pointer to the file to which code for expression is currently being written.

**ExpressionHeader**
PURPOSE: Boolean flag indicating whether an array declaration needs to be written before expression data structures can be written to the current *ExpressionFP* file.

**ExpressionVersion**
PURPOSE: An integer value representing the number of different *ExpressionFP* files that have been opened for the module being generated.

**FilePrefix**
PURPOSE: A pointer to the file name prefix string used to generate the file names for storing the C code (the first argument to `constructs-to-c`).

**HeaderFP**
PURPOSE: A pointer to the header file for the code module being generated.

**ImageID**
PURPOSE: Integer value containing the module ID number for the generated code (the second argument to `constructs-to-c`).

**ListOfCodeGeneratorItems**
PURPOSE: Contains a list of data structures used to call functions for various constructs which generate C code.
MaxIndices

PURPOSE: Integer value containing the preferred maximum number of array elements to store in a single file (the third argument to constructs-to-c).

GLOBAL FUNCTIONS

AddCodeGeneratorItem

PURPOSE: Adds an item to theListOfCodeGeneratorItems. Every construct that generates code for constructs-to-c must use this function to install functions that will be called whenever a constructs-to-c command is executed.

ARGUMENTS: A name to be associated with the code generator item, the priority of the item, a pointer to a function which is called before constructs-to-c begins generating code, a pointer to a function which is called for generating the C code data structures for the item, and a pointer to a function which is called to generate initialization C code for the InitCImage_<id> function.

ConstructsToCCommand

PURPOSE: Converts CLIPS constructs currently loaded into the CLIPS environment into C data structures and stores them in a file. This function is the driver routine for the CLIPS command constructs-to-c. The function GenerateCode is called by this function to perform the code generation.

ARGUMENTS: No actual arguments. The CLIPS arguments passing routines are used to extract arguments when this function is called from the CLIPS environment.

ConstructsToCCommandDefinition

PURPOSE: Sets up the definition of constructs-to-c command.

ExpressionToCode

PURPOSE: Prints a C code reference to an expression to a specified file and writes the C code representation of an expression to the current file storing expressions.

ARGUMENTS: A pointer to an open file and a pointer to an expression. The C code reference to the expression is written to the file pointer passed as a parameter and the C code
representation of the expression (to which the reference refers) is written to the file stored in the ExpressionFP global variable.

RETURNS: Integer value. 1 if the C code representation was successfully generated and written to the file, 0 if a NULL expression pointer was passed as an argument (in which case the C code representation is not written but the string “NULL” is written as the code reference), and -1 if the C code was not successfully generated.

OTHER NOTES: The function DumpExpression is called to write the C code representation of the expression.

NewCFile

PURPOSE: Opens a new file for writing C code.

ARGUMENTS: A pointer to the file name prefix string used to generate the file names for storing the C code, an integer suffix indicating the general contents of the file (e.g. 4 for defrules), and another integer suffix indicating the count of files for this type (e.g. the 5th file containing defrule information). For example, the file name “mab4_5.c” would be generated for the prefix string “mab” for the 5th file containing defrule information (assuming that 4 was the identification number for defrules).

RETURNS: A pointer to the newly opened file or NULL if the file could not be opened.

PrintDeffunctionReference

PURPOSE: Prints the C code representation of a pointer to a deffunction to a file.

ARGUMENTS: A pointer to an open file and a pointer to a deffunction.

PrintFloatReference

PURPOSE: Prints the C code representation of a FloatTable entry pointer to a file.

ARGUMENTS: A pointer to an open file and a pointer to a FloatTable entry.

PrintFunctionReference

PURPOSE: Prints the C code representation of a pointer to a function defined using DefineFunction.
ARGUMENTS: A pointer to an open file and a pointer to function.

**PrintGenericFunctionReference**

PURPOSE: Prints the C code representation of a pointer to a generic function to a file.

ARGUMENTS: A pointer to an open file and a pointer to a generic function.

**PrintIntegerReference**

PURPOSE: Prints the C code representation of an `IntegerTable` entry pointer to a file.

ARGUMENTS: A pointer to an open file and a pointer to a `IntegerTable` entry.

**PrintSymbolReference**

PURPOSE: Prints the C code representation of a `SymbolTable` entry pointer to a file.

ARGUMENTS: A pointer to an open file and a pointer to a `SymbolTable` entry.

**INTERNAL FUNCTIONS**

**DumpExpression**

PURPOSE: Prints a C code reference to an expression to a file.

ARGUMENTS: A pointer to an expression. The file the expression is written to is stored in the `ExpressionFP` global variable.

**FloatsToCode**

PURPOSE: Called by function `GenerateCode` to write the C code representation of all `FloatTable` entries to one or more files.

ARGUMENTS: A pointer to the file name prefix string used to generate the file names for storing the C code and an integer starting value for the file count ID. In generating file names, the name prefix is appended by the characters “1_” followed by an integer file count ID. The extension “.c” is then added to the file name.

RETURNS: Integer value. If the C code was successfully generated and written to the file(s), then the file count ID for the next file to
be generated is returned, otherwise zero is returned. If only one file is generated the value returned will be one greater than the file count ID passed as an argument.

### FunctionsToCode

**PURPOSE:** Called by function **GenerateCode** to write the C code representation of all functions defined using **DefineFunction** to one or more files.

**ARGUMENTS:** A pointer to the file name prefix string used to generate the file names for storing the C code. In generating file names, the name prefix is appended by the characters “2_” followed by an integer file count ID. The extension “.c” is then added to the file name.

**RETURNS:** Boolean value. TRUE, if the C code was successfully generated and written to the file(s), otherwise FALSE.

### GenerateCode

**PURPOSE:** Main driver routine for coordinating code generation for the **constructs-to-c** function.

**ARGUMENTS:** A pointer to the file name prefix string used to generate the file names for storing the C code (the first argument to **constructs-to-c**), the module ID number for the generated code (the second argument to **constructs-to-c**), and the preferred maximum number of array elements to store in a single file (the third argument to **constructs-to-c**).

**RETURNS:** Boolean value. TRUE, if the C code was successfully generated and written to the file(s), otherwise FALSE.

### HashTablesToCode

**PURPOSE:** Called by function **GenerateCode** to write the C code representation of the **IntegerTable**,** FloatTable**, and **SymbolTable** each to its own file.

**ARGUMENTS:** A pointer to the file name prefix string used to generate the file names for storing the C code. In generating file names, the name prefix is appended by the characters “1_” followed by an integer file count ID (1 for the **SymbolTable**, 2 for the **FloatTable**, and 3 for the **IntegerTable**). The extension “.c” is then added to the file name.

**RETURNS:** Boolean value. TRUE, if the C code was successfully generated and written to the files, otherwise FALSE.
**IntegersToCode**

**PURPOSE:** Called by function `GenerateCode` to write the C code representation of all `IntegerTable` entries to one or more files.

**ARGUMENTS:** A pointer to the file name prefix string used to generate the file names for storing the C code and an integer starting value for the file count ID. In generating file names, the name prefix is appended by the characters "1_" followed by an integer file count ID. The extension " .c" is then added to the file name.

**RETURNS:** Integer value. If the C code was successfully generated and written to the file(s), then the file count ID for the next file to be generated is returned, otherwise zero is returned. If only one file is generated the value returned will be one greater than the file count ID passed as an argument.

**ListUserFunctions**

**PURPOSE:** Writes C code extern declarations for all functions defined using `DefineFunction` in the header file for the construct module being generated.

**ARGUMENTS:** A file pointer to the header file for the construct module being generated.

**MarkBuckets**

**PURPOSE:** Replaces the bucket slot of each entry in the `SymbolTable`, `IntegerTable`, and `FloatTable` with an integer index that will be used to refer to that value. For example, the seventh symbol in the `SymbolTable` has the value 7 stored in its bucket slot (which would normally indicate the location in the `SymbolTable` that the symbol is stored).

**PrintCString**

**PURPOSE:** Prints the C code representation of a string replacing backslashes, quotation marks, and carriage returns with the appropriate character escape sequences.

**ARGUMENTS:** A pointer to an open file and a pointer to a string.

**SetUpInitFile**

**PURPOSE:** Writes the C code function needed for the initialization of a runtime module to a file. The function written is the
InitCImage_<id> function described in section 5 of the Advanced Programming Guide.

ARGUMENTS: A pointer to the file name prefix string used to generate the file name for storing the C code. In generating the file name, the name prefix is appended by a “.c” extension.

RETURNS: Boolean value. TRUE, if the C code was successfully generated and written to the file(s), otherwise FALSE.

SymbolsToCode

PURPOSE: Called by function GenerateCode to write the C code representation of all SymbolTable entries to one or more files.

ARGUMENTS: A pointer to the file name prefix string used to generate the file names for storing the C code and an integer starting value for the file count ID. In generating file names, the name prefix is appended by the characters “1_” followed by an integer file count ID. The extension “.c” is then added to the file name.

RETURNS: Integer value. If the C code was successfully generated and written to the file(s), then the file count ID for the next file to be generated is returned, otherwise zero is returned. If only one file is generated the value returned will be one greater than the file count ID passed as an argument.

UnmarkBuckets

PURPOSE: Restores the bucket slot of each entry in the SymbolTable, IntegerTable, and FloatTable with its appropriate value. This function is called to reverse the changes made by MarkBuckets.
Primary Functions Module

The Primary Functions Module (sysprime.c) provides a set of environment commands and procedural functions. Commands and functions provided are **watch**, **unwatch**, **clear**, **reset**, **exit**, **if**, **while**, **bind**, **progn**, **return**, and **break**. In addition, several functions for CLIPS internal use are defined in this module. These functions are **nop**, **constant**, **nonconstant**, **neq_field**, **eq_field**, **get_bind**, **pointer**, **get_field**, and **get_end**.

GLOBAL VARIABLES

<table>
<thead>
<tr>
<th>BreakContext</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: An integer flag indicating when the <strong>break</strong> function can be validly called. This flag is saved and cleared upon entry into a deffunction, generic function method or message-handler and restored on exit.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ReturnContext</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: An integer flag indicating when the <strong>return</strong> function can be validly called. This flag is saved and set upon entry into a deffunction, generic function method or message-handler and restored on exit.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ReturnFlag</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: An integer flag set when the <strong>return</strong> function is called and <strong>ReturnContext</strong> is set. The flag is cleared by the enclosing deffunction, generic function method or message-handler.</td>
</tr>
</tbody>
</table>

INTERNAL VARIABLES

<table>
<thead>
<tr>
<th>BreakFlag</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: An integer flag set when the <strong>break</strong> function is called and <strong>BreakContext</strong> is set. The flag is cleared by the enclosing <strong>while</strong> loop.</td>
</tr>
</tbody>
</table>

GLOBAL FUNCTIONS

<table>
<thead>
<tr>
<th>Primary Function Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Makes appropriate <strong>DefineFunction</strong> calls to notify CLIPS of functions defined in this module.</td>
</tr>
</tbody>
</table>
INTERNAL FUNCTIONS

Primary and Internal Functions

PURPOSE: A series of functions which defines the primary CLIPS RHS actions and internal functions listed above. See the Basic Programming Guide for more detail on individual functions.

OTHER NOTES: Some functionality for these functions is contained in other modules.
Predicate Functions Module

The Predicate Functions Module (syspred.c) provides a number of predicates and simple mathematical functions commonly used in CLIPS. Predicate functions provided are `eq`, `neq`, `symbolp`, `stringp`, `lexemep`, `integerp`, `floatp`, `numberp`, `oddp`, `evenp`, `multifieldp`, `pointerp`, and `or`, `not`, `=`, `<>`, `>`, `<`, `=>`, and `<=`. Mathematical functions provided are `*`, `/`, `+`, `-`, `div`, `set-auto-float-dividend`, and `get-auto-float-dividend`.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

<table>
<thead>
<tr>
<th>AutoFloatDividend</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>

GLOBAL FUNCTIONS

<table>
<thead>
<tr>
<th>PredicateFunctionDefinitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>

INTERNAL FUNCTIONS

<table>
<thead>
<tr>
<th>Predicate and Math Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>
The I/O Functions Module (sysio.c) provides a number of functions convenient for performing I/O. Among these are **open**, **close**, **read**, **readline**, **printout**, and **format**.

### GLOBAL VARIABLES

None.

### INTERNAL VARIABLES

None.

### GLOBAL FUNCTIONS

**IOFunctionDefinitions**

**PURPOSE:** Makes appropriate `DefineFunction` calls to notify CLIPS of functions defined in this module.

### LOCAL FUNCTIONS

**I/O Functions**

**PURPOSE:** A series of functions which defines I/O functions listed above. See the *Basic Programming Guide* for more detail on the individual functions.
Secondary Functions Module

The Secondary Functions Module (syssecnd.c) provides a set of useful functions that perform a wide variety of useful tasks. Functions provided are \texttt{trunc}, \texttt{integer}, \texttt{float}, \texttt{abs}, \texttt{min}, \texttt{max}, \texttt{str-assert}, \texttt{setgen}, \texttt{gensym}, \texttt{gensym*}, \texttt{system}, \texttt{length}, \texttt{time}, \texttt{random}, \texttt{seed}, \texttt{conserve-mem}, \texttt{release-mem}, \texttt{mem-used}, \texttt{mem-requests}, and \texttt{options}.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

<table>
<thead>
<tr>
<th>GensymNumber</th>
</tr>
</thead>
</table>

PURPOSE: An integer value used by \texttt{gensym} and \texttt{gensym*} in creating symbols.

GLOBAL FUNCTIONS

<table>
<thead>
<tr>
<th>SecondaryFunctionDefinitions</th>
</tr>
</thead>
</table>

PURPOSE: Makes appropriate \texttt{DefineFunction} calls to notify CLIPS of functions defined in this module.

INTERNAL FUNCTIONS

<table>
<thead>
<tr>
<th>Secondary Functions</th>
</tr>
</thead>
</table>

PURPOSE: A series of functions which defines secondary CLIPS RHS actions listed above. See the \textit{Basic Programming Guide} for more detail on individual functions.

OTHER NOTES: Some functionality for these functions is contained in other modules.
Multifield Functions Module

The Multifield Functions Module (multivar.c) provides a set of useful functions for use with multifield values. Functions provided are \texttt{nth}, \texttt{member}, \texttt{subset}, \texttt{mv-subseq}, \texttt{mv-delete}, \texttt{mv-append}, \texttt{mv-replace}, \texttt{str-explode}, and \texttt{str-implode}.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

None.

GLOBAL FUNCTIONS

\textbf{MultifieldFunctionDefinitions}

\textbf{PURPOSE:} Makes appropriate \texttt{DefineFunction} calls to notify CLIPS of functions defined in this module.

INTERNAL FUNCTIONS

\textbf{Multifield Functions}

\textbf{PURPOSE:} A series of functions which defines the CLIPS multifield functions listed above. See the Basic Programming Guide for more detail on individual functions.
String Functions Module

The String Functions Module (strings.c) provides a set of useful functions for manipulating strings. Functions provided are \texttt{str-length}, \texttt{str-compare}, \texttt{upcase}, \texttt{lowcase}, \texttt{sub-string}, \texttt{str-index}, \texttt{str-cat}, \texttt{sym-cat}, \texttt{eval}, and \texttt{build}.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

None.

GLOBAL FUNCTIONS

\textbf{StringFunctionDefinitions}

\textbf{PURPOSE:} Makes appropriate \texttt{DefineFunction} calls to notify CLIPS of functions defined in this module.

INTERNAL FUNCTIONS

\textbf{String Functions}

\textbf{PURPOSE:} A series of functions which define the CLIPS string functions listed above. See the \textit{Basic Programming Guide} for more detail on individual functions.
Math Functions Module

The Math Functions Module (math.c) provides a set of useful math functions beyond the basic math functions provided by the Predicate Functions Module. Functions provided are \( \cos \), \( \sin \), \( \tan \), \( \sec \), \( \csc \), \( \cot \), \( \arccos \), \( \arcsin \), \( \arctan \), \( \text{asec} \), \( \text{acsc} \), \( \text{acot} \), \( \cosh \), \( \sinh \), \( \tanh \), \( \text{sech} \), \( \text{csch} \), \( \text{coth} \), \( \text{acosh} \), \( \text{asinh} \), \( \text{atanh} \), \( \text{asech} \), \( \text{acsch} \), \( \text{acoth} \), \( \text{mod} \), \( \exp \), \( \log \), \( \log_{10} \), \( \sqrt{\ } \), \( \pi \), \( \text{deg-rad} \), \( \text{rad-deg} \), \( \text{deg-grad} \), \( \text{grad-deg} \), \( \text{**} \), and \( \text{round} \).

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

None.

GLOBAL FUNCTIONS

MathFunctionDefinitions

PURPOSE: Makes appropriate DefineFunction calls to notify CLIPS of functions defined in this module.

INTERNAL FUNCTIONS

Math Functions

PURPOSE: A series of functions which defines extended math functions listed above. See the Basic Programming Guide for more detail on individual functions.

OTHER NOTES: The Math Module does some checking to verify that illegal arguments are not passed to some functions since actions taken when an error occurs in these math functions can be machine dependent.
The Text Processing Module (textpro.c) provides a set of useful functions for building and accessing a hierarchical lookup system for multiple external files. Functions which provide on-line help are also available. The functions provided are **help**, **help-path**, **fetch**, **toss**, and **print-region**.

**GLOBAL VARIABLES**

None.

**INTERNAL VARIABLES**

None.

**GLOBAL FUNCTIONS**

<table>
<thead>
<tr>
<th>HelpFunctionDefinitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>

**INTERNAL FUNCTIONS**

<table>
<thead>
<tr>
<th>Text Processing Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>
File Commands Module

The File Commands Module (intrfile.c) provides a set of useful interface commands that performs certain file operations not associated with standard file I/O operations. The functions provided are **batch**, **load**, **save**, **bload**, **bsave**, **dribble-on**, **dribble-off**, **crsv-trace-on**, and **crsv-trace-off**.

GLOBAL VARIABLES

None.

INTERNAL VARIABLES

**Batch and Dribble Globals**

PURPOSE: Several variables containing information for batch and dribble commands.

GLOBAL FUNCTIONS

**FileFunctionDefinitions**

PURPOSE: Makes appropriate **DefineFunction** calls to notify CLIPS of commands defined in this module.

INTERNAL FUNCTIONS

**File Commands**

PURPOSE: A series of functions which define the file-oriented commands listed above. See the *Basic Programming Guide* for more detail on individual functions.

OTHER NOTES: Some functionality for these commands is provided in other modules. The load command is one example of this.
The Deffunction Module (defnctn.c) manages all aspects of the deffunction construct including parsing, execution, and removal. For a description of the deffunction construct, see the Basic Programming Guide. The deffunction construct capability can be removed by using the appropriate compile flag in the setup header file. The deffunction data structure is summarized in the following diagram:

<table>
<thead>
<tr>
<th>Name (Symbol Pointer)</th>
<th>Busy Count (int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Count (int)</td>
<td>Minimum Parameters (int)</td>
</tr>
<tr>
<td>Maximum Parameters (int)</td>
<td>Actions (Expression Pointer)</td>
</tr>
<tr>
<td>Pretty-Print Form (array of char)</td>
<td>Bload/Bsave Index (long int)</td>
</tr>
<tr>
<td>Previous Link (Deffunction Pointer)</td>
<td>Next Link (Deffunction Pointer)</td>
</tr>
</tbody>
</table>

The internal data structure of a deffunction construct primarily consists of: a symbolic name, two integers which indicate the minimum and maximum number of arguments the deffunction will accept respectively and a sequence of expressions which comprise the body of the deffunction. If a deffunction has a wildcard parameter (i.e. the deffunction will accept any number of arguments greater than or equal to the minimum number of arguments), the maximum number of arguments field will have the value -1. A busy count for each deffunction reflects how many other expressions in other constructs refer to that deffunction. This busy count must be zero before it is safe to delete the deffunction.

Similarly, an execution count for each deffunction reflects how many times a deffunction has been called. This execution count must be zero before a deffunction can be modified. Other fields in the deffunction data structure include: the pretty-print form, an index for use in binary load/save and the construct compiler, and pointers for double links to other deffunctions. A new deffunction is added to the list of deffunctions, DFList, before its actions are parsed so that it may be recursive, if desired.

When a deffunction is called, if the number of arguments is outside the acceptable range, the call is immediately terminated and an error is generated. Otherwise, all the actions of the deffunction are evaluated in order as if they were grouped in a progn. The evaluation of the last expression in the deffunction body is returned as the value of the deffunction, unless an error occurs or the return function is used (see the Basic Programming Guide).

Deffunction calls are represented by an expression data structure where the type field is PCALL (for procedure call) and the value field is the address of the corresponding deffunction construct. The expressions for the arguments of the deffunction are chained together via "next argument" pointers, and the whole chain is attached to the "argument list" pointer of the deffunction call. When such an expression is evaluated, the routines in the Evaluation Module call a special function,
**CallDeffunction**, to actually evaluate the arguments and perform the actions contained within the body of the deffunction.

The arguments of a deffunction are evaluated and stored in order in an array of data objects called the deffunction parameter array. Variable references within the body of a deffunction are replaced when the construct is loaded with function calls which either access the bind list (see the Primary Functions Module), get the value of a global variable (see the Defglobal Manager Module) or positionally access the deffunction parameter array. For example, references to the second parameter of a deffunction are replaced with a call to the function **DFRtnUnknown**, which accesses the second data object in the parameter array at run-time.

A wildcard parameter allows the deffunction to accept any number of arguments. All references to the wildcard parameter are replaced with a call to a special function, **DFWildargs**, which groups all of the data objects in the parameter array starting at the position of the wildcard parameter to the end of the array into a multifield data object.

If a parameter (including a wildcard parameter) is rebound anywhere within the body of the deffunction, all references to that parameter are replaced with calls to a special function, **DeffunctionGetBind**, which first checks the bind list before accessing the parameter array.

### GLOBAL VARIABLES

<table>
<thead>
<tr>
<th>deffunctionArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: A pointer to an array of deffunction data structures loaded using the <strong>bload</strong> command. When <strong>bload</strong> is in effect, <strong>DFList</strong> and <strong>DFBot</strong> will point to the first and last elements of this array respectively.</td>
</tr>
</tbody>
</table>

### INTERNAL VARIABLES

<table>
<thead>
<tr>
<th>CurrentDeffunctionName</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: A symbol indicating the name of the currently executing deffunction used for error and trace messages.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DeffunctionError</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: A flag used to indicate when deffunction parsing errors occur.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deffunction Trace Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: <strong>BEGIN_TRACE</strong> and <strong>END_TRACE</strong> are the strings used in trace printouts to indicate the beginning and end of execution of a deffunction.</td>
</tr>
</tbody>
</table>

OTHER NOTES: Implemented as preprocessor constants.
DFBot
PURPOSE: A pointer to the last node in the list of all currently defined deffunctions.

DFCount
PURPOSE: An intermediary counter used for deffunction data structures during binary loads and saves.

DFInputToken
PURPOSE: An intermediary variable used for scanned tokens by the deffunction parsing routines during a load.

DFList
PURPOSE: A pointer to the first node in the list of all currently defined deffunctions.

DFParamArray
PURPOSE: A pointer to an array of data objects which are the evaluated arguments for the currently executing deffunction.

DFParamSize
PURPOSE: An integer indicating the number of data objects in the currently executing deffunction's parameter array.

WatchDeffunctions
PURPOSE: An integer flag indicating whether or not to print out trace information whenever a deffunction begins and ends execution. This flag is used by the watch command.

GLOBAL FUNCTIONS

CallDeffunction
PURPOSE: This routine is called by EvaluateExpression in the Evaluation Module to process a deffunction call.

ARGUMENTS:
1) A pointer to a deffunction.
2) A list of expressions forming the deffunction arguments.
3) A pointer to a data object which will hold the return value of the deffunction.
OTHER NOTES: Following is a summary of CallDeffunction:

1. Count and check the number of arguments.
2. Save previous values of globals, such as CurrentDeffunctionName, and set them for the new deffunction.
3. Increment the evaluation depth (see the Evaluation Module).
4. Evaluate the arguments and store them in the deffunction parameter array.
5. Save the state of the bind list and then destroy it.
6. Save the states of the return and break contexts and set them to FALSE.
7. Increment the execution count of the deffunction.
8. Call EvaluateExpression for the actions of the deffunction and capture the result.
9. Restore all global values to their previous states.
10. Decrement the execution count of the deffunction.
11. Decrement the evaluation depth.
13. Adjust the evaluation depth of the return value (see PropogateReturnValue in the Evaluation Module).

### CmdListDeffunctions

**PURPOSE:** Lists all the currently defined deffunctions.

**OTHER NOTES:** Implementation of the CLIPS function list-deffunctions.

### CmdPPDeffunction

**PURPOSE:** Displays the pretty-print form of the deffunction specified by the CLIPS supplied argument.

**OTHER NOTES:** Implementation of the CLIPS function ppdeffunction.

### CmdUndeffunction

**PURPOSE:** Removes a deffunction.

**OTHER NOTES:** Implementation of the CLIPS function undefunction.

### DeffunctionGetBind

**PURPOSE:** Determines the value of a specified variable reference within the body of a deffunction. The symbolic name of the variable and an index indicating if the variable is a deffunction parameter are CLIPS supplied arguments. If the variable is
on the bind list, that value is returned. Otherwise, the value of
the parameter specified by the index is returned. In the event
that the variable is neither on the bind list nor a parameter,
an error will be generated.

ARGUMENTS: A pointer to a data object which will hold the value of the
bound variable.

OTHER NOTES: Implementation of the internal CLIPS function (df-getbind).

Used for general variable references, including bind list
variables and deffunction parameters which are rebound
within the actions of the deffunction. If the index is zero, the
variable is not a deffunction parameter. The absolute value
of the index minus one is the position of the parameter in the
deffunction parameter array. If the index is less than zero, the
variable corresponds to the wildcard parameter.

DFRtnUnknown

PURPOSE: Gets the value of the specified element of the deffunction
parameter array, where the element index plus one is given
as a CLIPS supplied argument.

ARGUMENTS: A pointer to a data object which will hold the value of the
bound variable.

OTHER NOTES: Implementation of the internal CLIPS function
(df-runknown).

Used for references to regular deffunction parameters which
are never rebound within the actions of the deffunction.

DFWildargs

PURPOSE: Gets the values of the specified elements of the deffunction
parameter array and groups them into a multifield data
object, where the range of elements is given by a CLIPS
supplied argument minus one to the end of the deffunction
parameter array.

ARGUMENTS: A pointer to a data object which will hold the value of the
bound variable.

OTHER NOTES: Implementation of the internal CLIPS function
(df-wildargs).

Used for references to a wildcard deffunction parameter
which is never rebound within the actions of the deffunction.
Embedded Access for Deffunctions

PURPOSE: The following functions are provided for embedded access and are documented in the Advanced Programming Guide: DeleteDeffunction, FindDeffunction, GetDeffunctionName, GetDeffunctionPPForm, GetNextDeffunction, IsDeffunctionDeletable and ListDeffunctions.

FindDeffunctionBySymbol

PURPOSE: Determines the address of a specified deffunction.
ARGUMENTS: A pointer to a symbol.
RETURNS: A pointer to a deffunction.

InitializeDeffunctions

PURPOSE: Defines all functions and commands for the deffunction construct. Sets up all necessary load, clear, save, watch, constructs-to-c and bload/bsave interfaces.
OTHER NOTES: Initialization differs between standard and run-time configurations.

SetListOfDeffunctions

PURPOSE: Initializes the global variables DFList and DFBot to point to the top and bottom respectively of the given list of deffunctions.
ARGUMENTS: A pointer to the top of a list of deffunctions.
OTHER NOTES: This function is used only in a run-time version of CLIPS.

INTERNAL FUNCTIONS

AddDeffunction

PURPOSE: Support routine for ParseDeffunction which allocates, initializes and attaches a new deffunction to the list of deffunctions.
ARGUMENTS: 1) The symbolic name of the new deffunction. 2) A list of expressions forming the actions of the deffunction. 3) The minimum number of parameters the deffunction will
accept.
4) The maximum number of parameters the deffunction will accept (-1 if the deffunction has a wildcard parameter).
5) An integer code indicating if the deffunction being added is a forward declaration (non-zero) or not (zero).

RETURNS: A pointer to the added deffunction.

**CheckDeffunctionCall**

PURPOSE: Determines if the number of CLIPS supplied arguments to a particular deffunction call is appropriate.

ARGUMENTS: 1) A pointer to the deffunction.
2) The number of arguments passed.

RETURNS: The integer zero for an incorrect number of arguments, non-zero otherwise.

**ClearDeffunctions**

PURPOSE: Used by the `clear` command to remove all currently defined deffunctions.

RETURNS: The integer zero if not all deffunctions were successfully cleared, non-zero otherwise.

OTHER NOTES: Deffunctions are removed after all other constructs except defgenerics, for this insures that the deffunctions are no longer in use. The use of priorities in `AddClearFunction` accomplishes this ordering.

**Deffunction Bload/Bsave Functions**

PURPOSE: A set of functions used by the `bload` and `bsave` commands to process the deffunction construct.

**Deffunction Constructs-To-C Functions**

PURPOSE: A set of functions used by the `constructs-to-c` command to process the deffunction construct.

**EvaluateDFParameters**

PURPOSE: Support routine for `CallDeffunction` which evaluates all the CLIPS supplied argument expressions for a deffunction
call and stores the resulting data objects in the deffunction parameter array (DFParamArray).

ARGUMENTS:  
1) The list of parameter name expressions.  
2) The number of parameters.

RETURNS:  
A pointer to an array of data objects containing the evaluations of the deffunction argument expressions.

FindParameter

PURPOSE:  
Support routine for ParseDeffunction used to determine if a parameter occurs more than once in a deffunction parameter list.

ARGUMENTS:  
1) The symbolic name of a parameter.  
2) The list of parameters parsed so far.

RETURNS:  
The integer zero if the named parameter is not already in the list, otherwise the position of the parameter in the list.

GrabWildargs

PURPOSE:  
Stores the deffunction parameter array elements from the specified beginning index minus one to the end of the array in the caller’s multifield data object.

ARGUMENTS:  
1) A pointer to a data object to hold the resulting multifield value.  
2) The index (one is the beginning) from which to start copying the parameter array.

ParseDeffunction

PURPOSE:  
Used by the load command to parse a deffunction.

ARGUMENTS:  
The logical name of the input source.

RETURNS:  
The integer zero if there are no parsing errors, non-zero otherwise.

ParseParameters

PURPOSE:  
Support routine for ParseDeffunction which parses a deffunction parameter list.

ARGUMENTS:  
1) The logical name of the input source.  
2) A buffer for holding a pointer to the symbolic name of a
wildcard parameter (if any).
3) A buffer for an integer code indicating any parsing errors.

RETURNS: A linked list of expressions containing the symbolic names of
the deffunction parameters.

---

**RemoveDeffunction**

PURPOSE: Removes a specified deffunction.

ARGUMENTS: A pointer to a deffunction.

---

**ReplaceParameters**

PURPOSE: Support routine for **ParseDeffunction** which replaces all
variable references in the deffunction actions with
appropriate function calls that access the bind list, the
deffunction parameter array or global variables at run-time.

ARGUMENTS: 1) The list of action expressions.
2) The list of parameter name expressions.
3) The symbolic name of a wildcard parameter (if any).

RETURNS: The integer zero if there are no parsing errors, non-zero
otherwise.

---

**SaveDeffunctions**

PURPOSE: Used by the **save** command to write out the pretty-print
forms of all the currently defined deffunctions.

ARGUMENTS: The logical name of the output destination.

---

**SaveDeffunctionHeaders**

PURPOSE: Used by the **save** command to write out forward
declarations of all deffunctions before other constructs in the
event that the deffunctions are called from these other
constructs.

ARGUMENTS: The logical name of the output destination.

---

**TraceDeffunction**

PURPOSE: Used by the **watch** command to print out trace messages
when a deffunction begins and ends execution.

ARGUMENTS: A string indicating the beginning or end of execution of a
deffunction.
The Generic Function Commands Module (genrccom.c) manages the parsing and general interface aspects of the defgeneric and defmethod constructs. For a description of the defgeneric and defmethod constructs, see the Basic Programming Guide. The generic function capability can be removed by using the appropriate compile flag in the setup header file.

The generic function data structures are summarized in the following diagram:

The internal data structure of a defgeneric construct primarily consists of: a symbolic name, an array of defmethod constructs (the explicit methods), a pointer to a
system or user-defined external function (if any), which is overloaded by the generic function (the implicit method) and an integer indicating the next available index for a new method. A busy count for each generic function reflects how many other expressions in other constructs refer to that generic function and how many times the generic function has been called. This busy count must be zero before it is safe to delete the generic function. Other fields in the generic function data structure include: the number of methods, the pretty-print form, an index for use in binary load/save and the construct compiler and pointers for double links to other generic functions. A new defgeneric is added to the list of generic functions, **GenericList**, before the actions of any its methods are parsed so that it may be recursive, if desired.

The internal data structure of a defmethod construct primarily consists of: two integers which indicate the minimum and maximum number of arguments the method will accept respectively, an array of parameter restriction data structures and a sequence of expressions which comprise the body of the method. Each parameter restriction data structure consists of: a sequence of classes, the number of classes in this sequence and a query expression. The corresponding run-time generic function argument must be an instance of one of these classes (if any) and the boolean query (if any) must be true in order for the restriction to be satisfied. If a method has a wildcard parameter (i.e. the method will accept any number of arguments greater than or equal to the minimum number of arguments), the maximum number of arguments field will have the value -1. Similarly, an execution count for each method reflects how many times a method has been called as well as how many outstanding generic function calls to which this method is applicable. This execution count must be zero before any methods for the generic function to which this method belongs can be modified. Other fields in the method data structure include: the pretty-print form and an identifying index.

As each new method is defined and inserted into the appropriate generic function's method array, the method array is maintained in sorted order according to precedence. This eases the burden on the generic dispatch at run-time. The Generic Function Functions Module covers method precedence in detail.

Generic function calls are represented by an expression data structure where the type field is GCALL and the value field is the address of the corresponding defgeneric construct. The expressions for the arguments of the generic function are chained together via "next argument" pointers, and the whole chain is attached to the "argument list" pointer of the generic function call. When such an expression is evaluated, the routines in the Evaluation Module call a special function, **GenericDispatch**, to actually evaluate the arguments and execute the method(s) of the generic function. The Generic Function Functions Module covers the generic dispatch in detail.

The arguments of a generic function are evaluated and stored in order in an array of data objects called the method parameter array. Variable references within the body of a defmethod are replaced when the construct is loaded with function calls which either access the bind list, get the value of a global variable or positionally access the method parameter array. For example, references to the second parameter of a method are replaced with calls to the function **RtnGenericUnknown** which access the second data object in the parameter array at run-time.

A wildcard parameter allows a method to accept any number of arguments. All references to the wildcard parameter are replaced with a call to a special function, **GetGenericWildargs**, which groups all of the data objects in the parameter array
starting at the position of the wildcard parameter to the end of the array into a multifield data object.

If a parameter (including a wildcard parameter) is rebound anywhere within the body of a method, all references to that parameter are replaced with calls to a special function, GetGenericBind, which first checks the bind list before accessing the parameter array.

**INTERNAL VARIABLES**

<table>
<thead>
<tr>
<th>GenericInputToken</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
<tr>
<td>An intermediary variable used for scanned tokens by the generic function parsing routines during a <strong>load</strong>.</td>
</tr>
</tbody>
</table>

**GLOBAL FUNCTIONS**

<table>
<thead>
<tr>
<th>CmdListDefgenerics</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
<tr>
<td>Lists all defgenerics in the system.</td>
</tr>
<tr>
<td>OTHER NOTES:</td>
</tr>
<tr>
<td>Implementation of the CLIPS function <strong>list-defgenerics</strong>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CmdListDefmethods</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
<tr>
<td>Lists the methods of the generic function(s) specified by the CLIPS supplied arguments.</td>
</tr>
<tr>
<td>OTHER NOTES:</td>
</tr>
<tr>
<td>Implementation of the CLIPS function <strong>list-defmethods</strong>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CmdUndefgeneric</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
<tr>
<td>Removes a generic function and all associated methods.</td>
</tr>
<tr>
<td>OTHER NOTES:</td>
</tr>
<tr>
<td>Implementation of the CLIPS function <strong>undefgeneric</strong>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CmdUndefmethod</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
<tr>
<td>Removes a generic function method.</td>
</tr>
<tr>
<td>OTHER NOTES:</td>
</tr>
<tr>
<td>Implementation of the CLIPS function <strong>undefmethod</strong>.</td>
</tr>
</tbody>
</table>

**Embedded Access for Generic Functions**

<table>
<thead>
<tr>
<th>PURPOSE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The following functions are provided for embedded access and are documented in the <em>Advanced Programming Guide</em>: <strong>DeleteDefgeneric</strong>, <strong>DeleteDefmethod</strong>, <strong>FindDefgeneric</strong>, <strong>GetNextDefgeneric</strong>, <strong>GetDefgeneric</strong>, <strong>GetDefmethod</strong>, <strong>InsertDefgeneric</strong>, <strong>InsertDefmethod</strong>, <strong>ListDefgenerics</strong>, <strong>ListDefmethods</strong>, <strong>Undefgeneric</strong>, <strong>Undefmethod</strong>.</td>
</tr>
</tbody>
</table>
GetDefgenericName, GetDefgenericPPForm, GetDefmethodDescription, GetNextDefmethod, GetDefmethodPPForm, IsDefgenericDeletable, IsDefmethodDeletable, ListDefgenerics and ListDefmethods.

### GetGenericBind

**PURPOSE:** Determines the value of a specified variable reference within the body of a method. The symbolic name of the variable and an index indicating if the variable is a method parameter are CLIPS supplied arguments. If the variable is on the bind list, that value is returned. Otherwise, the value of the parameter specified by the index is returned. In the event that the variable is neither on the bind list nor a parameter, an error will be generated.

**ARGUMENTS:** A pointer to a data object to hold the variable's value.

**OTHER NOTES:** Implementation of the internal CLIPS function *(gnrc-bind)*. Used for general variable references, including bind list variables and method parameters which are rebound within the actions of the method. If the index is zero, the variable is not a method parameter. The absolute value of the index minus one is the position of the parameter in the method parameter array. If the index is less than zero, the variable corresponds to the wildcard parameter.

### GetGenericWildargs

**PURPOSE:** Gets the values of the specified elements of the method parameter array and groups them into a multifield data object, where the range of elements is given by a CLIPS supplied argument minus one to the end of the method parameter array.

**ARGUMENTS:** A pointer to a data object to hold the variable's value.

**OTHER NOTES:** Implementation of the internal CLIPS function *(gnrc-wildargs)*. Used for references to a wildcard method parameter which is never rebound within the actions of the method.

### PPDefgeneric

**PURPOSE:** Displays the pretty-print form of the defgeneric specified by the CLIPS supplied argument.
OTHER NOTES: Implementation of the CLIPS function **ppdefgeneric**.

---

### PPDefmethod

**PURPOSE:** Displays the pretty-print form of the generic function method specified by the CLIPS supplied arguments.

**OTHER NOTES:** Implementation of the CLIPS function **ppdefmethod**.

---

### RtnGenericUnknown

**PURPOSE:** Gets the value of the specified element of the method parameter array, where the element index plus one is given as a CLIPS supplied argument.

**ARGUMENTS:** A pointer to a data object to hold the variable's value.

**OTHER NOTES:** Implementation of the internal CLIPS function (gnrc-runknown).

Used for references to regular method parameters which are never rebound within the actions of the method.

---

### SetupGenericFunctions

**PURPOSE:** Defines all functions and commands for the defgeneric and defmethod constructs. Sets up all necessary load, clear, save, watch, constructs-to-c and bload/bsave interfaces.

**OTHER NOTES:** Initialization differs between standard and run-time configurations.

---

###.TypeOf

**PURPOSE:** Determines the type (class) of the CLIPS supplied argument.

**ARGUMENTS:** A pointer to a data object to hold the symbolic name of the class of the CLIPS supplied argument.

**OTHER NOTES:** This function implements the CLIPS function type function when COOL is not available. When COOL is available, the functions type and class are implemented by GetInstanceClassCmd (see the Instance Commands Module).
INTERNAL FUNCTIONS

AddParameter

PURPOSE: Support routine for ParseParameters which links intermediate information for a method parameter and its restrictions to the list of other method parameters.

ARGUMENTS: 1) The top of the parameter expression list.
2) The bottom of the parameter expression list.
3) The parameter symbolic name.
4) A pointer to a parameter restriction data structure.

RETURNS: The (new) top of the parameter expression list.

CheckGenericExists

PURPOSE: Determines if a specified generic function exists.

ARGUMENTS: 1) The name of the calling function.
2) The name of the generic function.

RETURNS: A pointer to the generic function (NULL if not found).

CheckMethodExists

PURPOSE: Determines if a specified method of a generic function exists.

ARGUMENTS: 1) The name of the calling function.
2) A pointer to the generic function.
3) The index of the method.

RETURNS: The method array index (-1 if not found).

DeleteTempRestricts

PURPOSE: Support routine for ParseParameters which deallocates intermediate data structures used for method parameter restrictions.

ARGUMENTS: The list of parameter expressions.

DuplicateParameters

PURPOSE: Support routine for ParseParameters which determines if a method's parameter list contains any duplicate names.

ARGUMENTS: 1) The list of parameter name expressions.
2) Buffer for address of last node searched (can be used to
later attach new parameter).
3) The symbolic name of the parameter being checked.

RETURNS: A non-zero integer if duplicates are found, zero otherwise.

---

**FindParameter**

**PURPOSE:** Support routine for **ReplaceParameters** which determines the position of a particular parameter in the list of all method parameters.

**ARGUMENTS:**
1) The symbolic name of a parameter.
2) The list of parameters parsed so far.

**RETURNS:** The integer zero if the named parameter is not already in the list, otherwise the position of the parameter in the list.

---

**GrabGenericWildargs**

**PURPOSE:** Stores the method parameter array elements from the specified beginning index minus one to the end of the array in the caller’s multifield data object.

**ARGUMENTS:**
1) A pointer to a data object to hold the resulting multifield value.
2) The index (one is the beginning) from which to start copying the parameter array.

---

**PackRestrictionTypes**

**PURPOSE:** Support routine for **ParseRestriction** which packs the class restrictions for a method parameter into a contiguous array for easy reference.

**ARGUMENTS:**
1) The restriction data structure.
2) The types expression list

**OTHER NOTES:** If COOL is not present, then the types are integer codes representing the CLIPS types. -1 means all types are acceptable.

---

**ParseDefgeneric**

**PURPOSE:** Used by the **load** command to parse a defgeneric.

**ARGUMENTS:**
The logical name of the input source.

**RETURNS:** The integer zero if there are no parsing errors, non-zero otherwise.
**ParseDefmethod**

**PURPOSE:** Used by the `load` command to parse a defmethod.

**ARGUMENTS:** The logical name of the input source.

**RETURNS:** The integer zero if there are no parsing errors, non-zero otherwise.

---

**ParseMethodName**

**PURPOSE:** Support routine for `ParseDefgeneric` which parses a generic function name.

**ARGUMENTS:** The logical name of the input source.

**RETURNS:** The symbolic name of the method.

---

**ParseMethodNameAndIndex**

**PURPOSE:** Support routine for `ParseDefmethod` which parses a method name and optional index.

**ARGUMENTS:** 1) The logical name of the input source. 2) Buffer for method index (0 if not specified).

**RETURNS:** The symbolic name of the method.

---

**ParseParameters**

**PURPOSE:** Support routine for `ParseDefmethod` which parses a method parameter list.

**ARGUMENTS:** 1) The logical name of the input source. 2) Buffer for the parameter name list. 3) Buffer for wildcard symbol (if any).

**RETURNS:** The number of parameters, or -1 on errors.

---

**ParseRestriction**

**PURPOSE:** Support routine for `ParseParameters` which parses the restrictions for a given method parameter.

**ARGUMENTS:** The logical name of the input source.

**RETURNS:** A pointer to a parameter restriction data structure, NULL on errors.
RemoveGeneric

PURPOSE: Removes a specified generic function and all its methods.

ARGUMENTS: A pointer to the generic function.

RETURNS: The integer one if successful, zero otherwise.

RemoveGenericMethod

PURPOSE: Removes a specified method of a generic function.

ARGUMENTS: 1) A pointer to the generic function.
2) The array index of the method.

ReplaceParameters

PURPOSE: Support routine for ParseDefmethod which replaces all variable references in the method actions with appropriate function calls that access the bind list, the method parameter array or global variables at run-time.

ARGUMENTS: 1) The list of action expressions.
2) The list of parameter name expressions.
3) The symbolic name of a wildcard parameter (if any).

RETURNS: The integer zero if there are no parsing errors, non-zero otherwise.

SaveDefgenerics

PURPOSE: Used by the save command to write out forward declarations of all generic functions before other constructs in the event that the generic functions are called from these other constructs.

ARGUMENTS: The logical name of the output destination.

SaveDefmethods

PURPOSE: Used by the save command to write out the pretty-print forms of all the methods of currently defined generic functions.

ARGUMENTS: The logical name of the output destination.
ValidType

PURPOSE: Support routine for ParseRestriction which determines if a class restriction list for a method parameter is comprised of existing classes.

ARGUMENTS: The symbolic name of the restriction class.

RETURNS: When COOL is present, an expression containing a pointer to the class, otherwise an expression list containing the integer codes of the CLIPS types (see constant.h) corresponding to the class, NULL on errors.
Generic Function Functions Module

The Generic Function Functions Module (genrcfun.c) establishes the precedence between different methods of a generic function when they are defined, manages the generic dispatch and provides support routines for other internal manipulations of generic functions and methods, such as allocation and deletion. For a description of the defgeneric and defmethod constructs, see the Basic Programming Guide. The generic function capability can be removed by using the appropriate compile flag in the setup header file.

Whenever a new method for a generic function is defined, the method array for that generic function is reallocated to make room for the new method information. The new method is inserted into the array such that a sorted order according to precedence is maintained. Section 8.5.2 in the Basic Programming Guide explains the method precedence rules in detail. The precedence between any two methods is determined by comparing field per field the parameter restrictions of the two methods.

When a generic function is called, CLIPS uses the generic function's arguments to find and execute the appropriate method. This process is termed the generic dispatch. The generic dispatch first forms a list of all the applicable methods to the generic function call. The methods in this list are linked using a temporary data structure called a method link:

```
Method Link       Method (Method Pointer)
                  Next Link (Method Link Pointer)
```

The first field is a pointer to an applicable method, and the second is a pointer to another method link.

For every method in the method array of the generic function, the parameter restriction list is checked against the actual arguments.

If the number of arguments is outside the acceptable range, the call is immediately terminated and an error is generated. Otherwise, all the actions of the method are evaluated in order as if they were grouped in a progn. The evaluation of the last expression in the deffunction body is returned as the value of the deffunction, unless an error occurs or the return function is used (see the Basic Programming Guide). The mechanics of a generic dispatch are outlined in the description of the function GenericDispatch.

GLOBAL VARIABLES

<table>
<thead>
<tr>
<th>CurrentGeneric</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: A pointer to the currently executing generic function.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CurrentMethod</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: A pointer to the currently executing method.</td>
</tr>
</tbody>
</table>
**GenericList**

PURPOSE: A pointer to the first node in the list of all currently defined generic functions.

**GenericListBottom**

PURPOSE: A pointer to the last node in the list of all currently defined generic functions.

**GenericStackFrame**

PURPOSE: A pointer to an array of expressions which are the evaluated arguments for the currently executing generic function. This variable is also referred to as the method parameter array.

OTHER NOTES: The method parameter array is stored as an array of expressions rather than data objects so that implicit methods (i.e. system functions) can easily be called with these arguments.

**GenericStackSize**

PURPOSE: An integer indicating the number of data objects in the currently executing generic function's method parameter array.

**WatchGenerics**

PURPOSE: An integer flag indicating whether or not to print out trace information whenever a generic function begins and ends execution. This flag is used by the `watch` command.

**WatchMethods**

PURPOSE: An integer flag indicating whether or not to print out trace information whenever an individual method begins and ends execution. This flag is used by the `watch` command.

**INTERNAL VARIABLES**

**Generic Function Trace Codes**

PURPOSE: `SYSTEM_NO` and `SYSTEM_YES` are integer codes used in trace printouts to indicate whether the method is an explicit defmethod or a system function.

OTHER NOTES: Implemented as preprocessor constants.
Generic Function Trace Strings

**PURPOSE:** BEGIN_TRACE and END_TRACE are the strings used in trace printouts to indicate the beginning and end of execution of a generic function or a method.

**OTHER NOTES:** Implemented as preprocessor constants.

Method Precedence Codes

**PURPOSE:** Three integer codes are used by FindMethodByRestrictions to indicate the relative precedence between two methods: HIGHER_PRECEDENCE, IDENTICAL and LOWER_PRECEDENCE.

**OTHER NOTES:** Implemented as preprocessor constants.

NextInCore

**PURPOSE:** A method link to the method shadowed by the currently executing method.

OldGenericBusySave

**PURPOSE:** An integer variable used to preserve the busy count of a generic function when a new method is added or deleted. Methods which recursively call the generic function to which they apply do not increment the generic function's busy count. This makes it possible to tell when it is safe to delete a generic function and its methods (i.e. when no other constructs refer to the generic function and none of the generic function's methods are executing).

TopOfCore

**PURPOSE:** The first method link of a linked list of methods which are applicable to the current generic function call. The list is in order according to method precedence.

GLOBAL FUNCTIONS

AddGeneric

**PURPOSE:** Support routine for ParseDefgeneric and ParseDefmethod in the Generic Function Commands Module. Adds a new generic function header to the list of generic functions.
ARGUMENTS: 1) Symbolic name of the new generic function.
2) Buffer for flag indicating if generic function is new or not.

RETURNS: A pointer to the (new) generic function.

---

**AddMethod**

PURPOSE: Support routine for **ParseDefmethod** in the Generic Function Commands Module. Stores all parsed information for a new method in the method array for the generic function.

ARGUMENTS: 1) A pointer to the generic function.
2) Old method address (can be NULL).
3) Old method array position (can be -1).
4) Method index to assign (0 if don’t care).
5) Parameter expression-list.
6) The number of parameters.
7) The wildcard symbol (NULL if none).
8) Method action expressions.
9) Method pretty-print form.

RETURNS: A pointer to the (new) method.

---

**CallNextMethod**

PURPOSE: Executes a method shadowed by the currently executing method. This function can only be called from the actions of currently executing method.

ARGUMENTS: A pointer to a data object to store the return value of the shadowed method.

OTHER NOTES: Following is a summary of **CallNextMethod**:

1*. Save the state of the bind list and then destroy it.
2. If an explicit shadowed method (see **NextInCore**) is not available, go to step 4.
3. Call **EvaluateExpression** for the actions of shadowed method, capture result and go to step 5.
4. If there is an implicit method, call it with **EvaluateExpression**, and capture the result.
5. Clear **ReturnFlag**.
6*. Restore the previous bind list.

*A bug exists in CLIPS version 5.1 that allows shadowed methods to affect locally bound variables of methods which are shadowing them. This is because Steps 1 and 6 are not present in the CLIPS 5.1 implementation of
CallNextMethod.

Implementation of the CLIPS function **call-next-method**.

---

### ClearDefgenerics

**PURPOSE:** Used by the `clear` command to remove all currently defined generic functions.

**RETURNS:** The integer zero if not all generic functions and methods were successfully cleared, non-zero otherwise.

**OTHER NOTES:** Methods are removed before other constructs which may use generic functions, for this insures that those constructs are no longer in use by any methods. Generic functions are cleared after the other constructs to insure that they are no longer in use by the other constructs. The use of priorities in `AddClearFunction` accomplishes this ordering.

---

### ClearDefmethods

**PURPOSE:** Used by the `clear` command to remove all currently defined generic function methods.

**RETURNS:** The integer zero if not all methods were successfully cleared, non-zero otherwise.

---

### DeleteMethodInfo

**PURPOSE:** Deallocates internal data structures associated with a method but does not remove the method from the generic function’s method array.

**ARGUMENTS:**

1) A pointer to a generic function.
2) A pointer to a method.

---

### FindDefgenericBySymbol

**PURPOSE:** Determines the address of a specified generic function.

**ARGUMENTS:** A pointer to a symbol.

**RETURNS:** A pointer to a generic function.

---

### FindMethodByIndex

**PURPOSE:** Support routine for `ParseDefmethod` in the Generic Function Commands Module. Determines if a method of the specified index already exists for the generic function.
ARGUMENTS: 1) A pointer to a generic function.
2) A method index.

RETURNS: The position of the method in the generic function's method array, -1 if not found.

---

**FindMethodByRestrictions**

PURPOSE: Support routine for `ParseDefmethod` in the Generic Function Commands Module. Examines the parsed parameter restrictions for the new method and determines if a method with matching parameter restrictions already exists for the generic function.

ARGUMENTS: 1) A pointer to a generic function.
2) Parameter expression list.
3) Number of parameters.
4) Wildcard symbol (can be NULL).
5) Buffer for holding array position of where to add new method (-1 if method already present).

RETURNS: A pointer to the method if found, NULL otherwise.

---

**GenericDispatch**

PURPOSE: This routine is called by `EvaluateExpression` in the Evaluation Module to process a generic function call.

ARGUMENTS: 1) A pointer to a generic function.
2) A list of expressions forming the generic function arguments.
3) A pointer to a data object which will hold the return value of the generic function.

OTHER NOTES: Following is a summary of `GenericDispatch`:

1. Save previous values of globals, such as `CurrentGeneric`, and set them for the new generic function.
2. Increment the evaluation depth (see the Evaluation Module).
3. Count and evaluate the arguments and store them in the method parameter array.
4. Save the state of the bind list and then destroy it.
5. Save the states of the `return` and `break` contexts and set them to FALSE.
6. Increment the busy count of the generic function.
7. Determine the set of applicable explicit methods (see `FindApplicableMethods`). If there are no applicable
explicit methods, go to step 9.
8. Call \texttt{EvaluateExpression} for the actions of explicit
method with the highest precedence, capture the result and
9. If there is an implicit method, call it with
\texttt{EvaluateExpression}, capture the result and go to step 11.
10. Generate an error indicating that there are no applicable
methods for the generic function call.
11. Restore all global values to their previous states.
12. Decrement the execution count of the deffunction.
13. Decrement the evaluation depth.
14. Clear \texttt{ReturnFlag}.
15. Adjust the evaluation depth of the return value (see
\texttt{PropogateReturnValue} in the Evaluation Module).

\begin{enumerate}
\item \textbf{MethodAlterError}
\begin{itemize}
\item \textbf{PURPOSE:} Displays an error message when an attempt is made to
modify an executing method.
\item \textbf{ARGUMENTS:} The name of the generic function.
\end{itemize}
\item \textbf{MethodsExecuting}
\begin{itemize}
\item \textbf{PURPOSE:} Determines if any of the methods of a generic function are
currently executing.
\item \textbf{ARGUMENTS:} A pointer to a generic function.
\item \textbf{RETURNS:} The integer zero if no methods are executing, non-zero
otherwise.
\end{itemize}
\item \textbf{NextMethodP}
\begin{itemize}
\item \textbf{PURPOSE:} Determines if a shadowed generic function method is
available for execution.
\item \textbf{RETURNS:} The integer zero if there is no method available, non-zero
otherwise.
\item \textbf{OTHER NOTES:} Implementation of the CLIPS function \texttt{next-methodp}.
\end{itemize}
\item \textbf{PreviewGeneric}
\begin{itemize}
\item \textbf{PURPOSE:} Displays all the applicable methods for a particular generic
function call. The generic function and arguments are
supplied by CLIPS.
\end{itemize}
\end{enumerate}
OTHER NOTES: Implementation of the CLIPS function `preview-generic`.

PrintMethod

PURPOSE: Support routine for use with trace messages and debugging displays which lists a brief description of the parameter restrictions for a method.

ARGUMENTS: 1) Buffer for method text description. 2) Buffer size (not including space for null character). 3) A pointer to a method.

SetGenericList

PURPOSE: Initializes the global variables `GenericList` and `GenericListBottom` to point to the top and bottom respectively of the given list of generic functions.

ARGUMENTS: A pointer to the top of a list of generic functions.

OTHER NOTES: This function is used only in a run-time version of CLIPS.

INTERNAL FUNCTIONS

AddGenericMethod

PURPOSE: Support routine for `AddMethod` which inserts an initialized method into the specified position of the method array.

ARGUMENTS: 1) A pointer to a generic function. 2) Position in the method array to add the new method. 3) The method index (0 if don't care).

RETURNS: A pointer to the new method.

DestroyMethodLinks

PURPOSE: Deallocates the linked list of applicable methods for a particular generic function call.

ARGUMENTS: A method link.

DetermineRestrictionClass

PURPOSE: Support routine for `IsMethodApplicable` which determines the class of a particular generic function argument.
ARGUMENTS: A pointer to an expression.

RETURNS: A pointer to the class of the argument, or NULL on errors.

OTHER NOTES: This function is used only when COOL is present. When COOL is not present, the CLIPS type integer codes found in constant.h are used in lieu of classes.

DisplayGenericCore

PURPOSE: Support routine for PreviewGeneric which displays the linked list of applicable methods for a particular generic function call.

ARGUMENTS: 1) A pointer to the generic function.
2) A method link for the list of applicable methods.

EvaluateGenericParameters

PURPOSE: Support routine for GenericDispatch which evaluates all the CLIPS supplied argument expressions for a generic function call and stores the resulting values in the method parameter array (GenericStackFrame).

ARGUMENTS: 1) The list of parameter name expressions.
2) The number of parameters.

RETURNS: A pointer to an array of expressions containing the evaluations of the generic function argument expressions.

FindApplicableMethods

PURPOSE: Support routine for GenericDispatch which determines the set of methods which are applicable to a particular generic function call.

ARGUMENTS: A pointer to a generic function.

RETURNS: A series of method links (TopOfCore), ranked according to precedence, which are applicable to the generic function call.

OTHER NOTES: The method array of the generic function is examined in order, and each method that has parameter restrictions which are satisfied by the generic function arguments are attached to the end of a list of applicable methods. Since the method array is in order according to precedence, the final list of applicable methods is automatically ranked appropriately.
**IsMethodApplicable**

**PURPOSE:** Support routine for **FindApplicableMethods** which determines if the parameter restrictions of a particular method are satisfied by the generic function arguments.

**ARGUMENTS:** A pointer to the method.

**RETURNS:** The integer zero if the method is not applicable, non-zero otherwise.

**NewGeneric**

**PURPOSE:** Support routine for **AddGeneric** which allocates and initializes a new generic function.

**RETURNS:** A pointer to an initialized generic function.

**RestoreBusyCount**

**PURPOSE:** Uses the internal variable **OldGenericBusySave** to restore the busy count of a generic function.

**ARGUMENTS:** A pointer to a generic function.

**OTHER NOTES:** Implemented as a preprocessor macro.

**RestrictionsCompare**

**PURPOSE:** Support routine for **FindMethodByRestrictions** which compares a new restriction expression list for the method currently being parsed with the parameter restrictions of an existing method to determine which set of restrictions has higher precedence.

**ARGUMENTS:**
1) The parameter restriction expression list.
2) The number of minimum restrictions.
3) The number of maximum restrictions (-1 if unlimited).
4) A pointer to a method with which to compare restrictions.

**RETURNS:** An integer code indicating the precedence between the two restriction sets:
-1: New restrictions have higher precedence.
0: New restrictions are identical.
1: New restrictions have lower precedence.
### SaveBusyCount

**PURPOSE:** Uses the internal variable **OldGenericBusySave** to save the busy count of a generic function.

**ARGUMENTS:** A pointer to a generic function.

**OTHER NOTES:** Implemented as a preprocessor macro.

### TraceGeneric

**PURPOSE:** Used by the **watch** command to print out trace messages when a generic function begins and ends execution.

**ARGUMENTS:** A string indicating the beginning or end of execution of a generic function.

### TraceMethod

**PURPOSE:** Used by the **watch** command to print out trace messages when a method begins and ends execution.

**ARGUMENTS:**
1) A string indicating the beginning or end of execution of a method.
2) A flag indicating whether the method being traced is an explicit or implicit method.

### TypeListCompare

**PURPOSE:** Support routine for **RestrictionsCompare** which determines the precedence between the class lists on two parameter restrictions.

**ARGUMENTS:**
1) A pointer to the first restriction data structure.
2) A pointer to the second restriction data structure.

**RETURNS:** An integer code indicating the precedence between the two parameter restrictions’ class lists:
-1: First restriction class list precedes the second.
0: Restriction class lists are identical.
1: Second restriction class list precedes the first.
The Generic Function Construct Compiler Interface Module (genrccmp.c) provides the functionality for implementing the constructs-to-c functions for the defgeneric and defmethod constructs.
The Generic Function Binary Load/Save Interface Module (genrcbin.c) provides the functionality for implementing the \texttt{bload} and \texttt{bsave} functions for the \texttt{defgeneric} and \texttt{defmethod} constructs.

Generic function methods can contain pointers to \texttt{defclasses} in their restrictions. To insure that the binary save files are identical whether or not COOL is present for methods which use only the system-defined primitive type classes, integer codes are used to represent the classes. If COOL is not present, these integer codes correspond to the CLIPS type codes found in \texttt{constant.h}. Otherwise, these integer codes are indices into an array of \texttt{defclasses}. The primitive type classes are stored in the same order as reflected in the values of the codes in \texttt{constant.h} (see the description of \texttt{ClassList} in the Class Functions Module).
The Class Commands Module (classcom.c) manages the parsing and general interface aspects of the defclass construct. For a description of the defclass construct, see the Basic Programming Guide. The defclass construct capability, along with the other features of the CLIPS Object-Oriented Language (COOL), can be removed by using the appropriate compile flag in the setup header file.

The class data structures are summarized in the following diagrams:

<table>
<thead>
<tr>
<th>Name (Symbol Pointer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash Value (int)</td>
</tr>
<tr>
<td>Binary Load/Save Index (long int)</td>
</tr>
<tr>
<td>Busy Count (int)</td>
</tr>
<tr>
<td>Predecessor Count (int)</td>
</tr>
<tr>
<td>Pretty-Print Form (array of char)</td>
</tr>
<tr>
<td>Traversal IDs (array of char)</td>
</tr>
<tr>
<td>Instance List (Instance Pointer)</td>
</tr>
<tr>
<td>Bitfields</td>
</tr>
<tr>
<td>Inheritance Links</td>
</tr>
<tr>
<td>Class List and Hash Table Links</td>
</tr>
<tr>
<td>Slot Information</td>
</tr>
<tr>
<td>Handler Information</td>
</tr>
</tbody>
</table>

The last five boxes in italics are expanded in further diagrams.

The internal data structure of a defclass construct consists primarily of: a symbolic name, a pointer to the first direct instance (see the Instance Commands Module for details on the instance data structures), links to the superclasses, subclasses and class precedence list of the class, an array of slot descriptors, a template of slot pointers available to direct instances and an array of message-handlers. A busy count for each class reflects how many other expressions in other constructs refer to that class, how many direct instances of the class exist and how many times the class is in use by various other COOL access functions. This busy count must be zero for a class and all its subclasses before it is safe to delete the class. Other fields in the defclass data structure include: a hash value indicating the position of the class in the hash table, a predecessor class count for use in determining the precedence list for a class and its subclasses, a bitmap where each bit corresponds to a unique class hierarchy traversal, a series of bitfields indicating such things as whether the class is a system class, links connecting the class to the global class list and hash table, the pretty-print form and an index for use in binary load/save and the construct compiler.

The traversal id bitmap is an array of TRAVERSAL_BYTES (see the Instance-Set Queries Module) characters. The number of bits in this map indicate how many simultaneous class hierarchy traversals can examine a class at once. Many of
the COOL access routines use recursive descent to access the subclasses of a class, e.g. instances when listing the indirect instances of a class. Due to multiple inheritance, it is possible for a class to be reached more than once via a straightforward recursive descent on subclasses. Thus, it is necessary to mark classes once they have been visited so that that branch of the search will not be repeated. However, since there may be more than one class hierarchy traversal occurring at a time (e.g. nested instance-set query functions), it is necessary to have unique markers for each class per traversal. When a class hierarchy traversal begins, a unique traversal id is requested via GetTraversalID. The bit corresponding to this id is cleared in the traversal maps of all classes. When a traversal comes to a class, it first checks the traversal bitmap against its traversal id with TestTraversalID. If the bit is already set, then it is known that this branch has already been explored. Otherwise, SetTraversalID is used to mark the class, and the traversal continues downwards. When a traversal is complete, ReleaseTraversalID makes the id available for use for another traversal. All of the variables, constants and functions dealing with traversal ids can be found in the Instance-Set Queries Module.

The bitfields are stored in a single integer and indicate the following information about a class: whether all the atoms and construct references within the defclass have had their busy counts incremented (i.e. whether the class has been installed), whether the class is a predefined system class, whether the class can have direct instances, whether the class corresponds to one of the primitive types defined in constant.h (SYMBOL, INTEGER, etc.) and the primitive type code for the class if the "primitive" bitfield is set.

The inheritance links are lists of the direct superclasses, direct subclasses and inheritance precedence list of a class. All three of these lists are formed using an intermediary data structure called a class link. These lists are not formed using direct
class pointers in the defclass itself because a particular class can be a superclass or subclass (direct or indirect) of many different classes.

The first field of this data structure is a pointer to a defclass, and the second is a pointer to another link. The three inheritance links in a defclass are all class link pointers. The inheritance links are built when the defclass is parsed (see ParseDefclass). In particular, the precedence list is formed by FindPrecedenceList, which is explained in detail in the Class Functions Module.

Unlike the inheritance links, the links which place a class in the global class list (ClassList) and class hash table (ClassTable) are implemented with direct class pointers in the defclass. This is because these links are unique to a class.

The slot information for a class is comprised of: an array of slot descriptors, which includes information for all the slots directly defined in a class; the number of slot descriptors; a template of all the slots which will be present in instances of a class, including slots directly inherited from the class and indirectly inherited from superclasses; the number of slots in the template; and an array of integer indices into the instance template which gives the order of the template slots according to the hash value of the symbolic names of the slots. The instance template is a contiguous array of slot descriptor pointers sorted by inheritance from least specific to most specific (slots from the same class are in the order they appeared in the defclass). The sorted
template map allows an instance slot to be found easily by performing a binary search on the symbolic hash value of the slot name. All direct instances of a class share the same instance template; each instance only needs to have its own array of slot values (see the general notes in the Instance Commands Module).

<table>
<thead>
<tr>
<th>Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Class</td>
<td>(Class Pointer)</td>
</tr>
<tr>
<td>Name</td>
<td>(Symbol Pointer)</td>
</tr>
<tr>
<td>Read Slot-Accessor</td>
<td>(Symbol Pointer)</td>
</tr>
<tr>
<td>Write Slot-Accessor</td>
<td>(Symbol Pointer)</td>
</tr>
<tr>
<td>Bitfields</td>
<td>(all stored in 1 integer)</td>
</tr>
<tr>
<td>Default Value</td>
<td>(Expression Pointer)</td>
</tr>
<tr>
<td>Shared Value</td>
<td>(Data Object Pointer)</td>
</tr>
<tr>
<td>Number of</td>
<td>Instances w/ Shared Slot (int)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each slot descriptor is comprised of the following: a pointer to the class in which this slot is directly defined, a symbolic name, a symbolic name of a read slot-accessor of the form `get-<slot-name>`, a symbolic name of a write slot-accessor of the form `put-<slot-name>`, a series of bitfields indicating the facets, an expression which yields the default value for a slot when evaluated during a `make-instance` call, a pointer to the data object holding the value of a shared slot at run-time and the number of instances of referencing a shared slot. If a slot is not shared, the last two fields will always be NULL and zero respectively. Local slot values are stored with the instances (see the general notes in the Instance Commands Module). The count of the number of instances sharing a slot is used to determine when to automatically initialize or erase a shared slot.

The bitfields are stored in a single integer and indicate the following information about a slot descriptor: whether a cardinality facet (`single` or `multiple`) was specified in the defclass, whether a storage facet (`shared` or `local`) was specified in the defclass, whether an access facet (`read-write`, `read-only` and `initialize-only`) was specified in the defclass, whether a default value facet (`default` or `default-dynamic`) was specified in the defclass, whether the default value (if any) is dynamic, whether the slot has shared or local storage, whether the slot has single or multiple cardinality,
whether the slot gets facets exclusively from the direct parent class or compositely from indirect superclasses as well, whether the slot is propagated to subclasses, whether the slot can be written, whether the slot can be written only during initialization of an instance and whether a slot has a default value override during a **make-instance** call.

```
<table>
<thead>
<tr>
<th>Handler Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handler Count (int)</td>
</tr>
<tr>
<td>Hash-Value Sorted Handler Map (array of ints)</td>
</tr>
</tbody>
</table>
```

The message-handler information for a class is comprised of: an array of handlers, the number of handlers and an array of integer indices into the handler array which gives the order of the handlers according to the hash value of their symbolic names. The sorted handler map allows a handler to be found easily by performing a binary search on the symbolic hash value of the handler name. For details on the message-handler data structure, see the Message-Handler Commands Module.

### GLOBAL VARIABLES

**ObjectParseToken**

**PURPOSE:** An intermediary variable used for scanned tokens by the COOL parsing routines during a **load**.

### INTERNAL VARIABLES

**Defclass Constants**

**PURPOSE:** The following are constants used in to determine when qualifiers in the defclass construct are duplicated: `CARDINALITY_BIT`, `STORAGE_BIT`, `ACCESS_BIT`, `INHERIT_BIT` and `COMPOSITE_BIT`.

**OTHER NOTES:** Implemented as preprocessor constants.

**Defclass Keywords**

**PURPOSE:** The following are keywords used in parsing a defclass: `SUPERCLASS_RLN`, `ABSTRACT_RLN`, `CONCRETE_RLN`, `HANDLER_DECL`, `SLOT_RLN`, `SLOT_DEF_RLN`, `SLOT_DEF_DYN_RLN`, `SLOT_NOINH_RLN`, `SLOT_INH_RLN`, `SLOT_RDONLY_RLN`, `SLOT_RDWRT_RLN`, `SLOT_RDONLY_RLN`, `SLOT_RDWRT_RLN`, `SLOT_RDONLY_RLN`. 
SLOT_SHARE_RLN, SLOT_LOCAL_RLN, SLOT_MULT_RLN, SLOT_SGL_RLN, SLOT_INIT_RLN, SLOT_COMPOSITE_RLN and SLOT_EXCLUSIVE_RLN.

OTHER NOTES: Implemented as preprocessor constants.

GLOBAL FUNCTIONS

### BrowseClassesCmd

**PURPOSE:** Displays an inheritance "graph" of the subclasses of defclass specified by the CLIPS supplied argument.

**OTHER NOTES:** Implementation of the CLIPS function `browse-classes`.

### ClassHandlersCmd

**PURPOSE:** Groups the message-handler names of a class specified by the CLIPS supplied argument into a multifield variable.

**ARGUMENTS:** A pointer to a data object to hold the resulting multifield.

**OTHER NOTES:** Implementation of the CLIPS function `class-message-handlers`.

### ClassHasHandler

**PURPOSE:** Determines if a message-handler is present in a class. Both arguments are supplied by CLIPS.

**RETURNS:** A non-zero integer if the message-handler is present in the class, zero otherwise.

**OTHER NOTES:** Implementation of the CLIPS function `class-message-handler-existp`.

### ClassHasSlot

**PURPOSE:** Determines if a slot is present in a class. Both arguments are supplied by CLIPS.

**RETURNS:** A non-zero integer if the slot is present in the class, zero otherwise.

**OTHER NOTES:** Implementation of the CLIPS function `class-slot-existp`. 
### ClassSlotsCmd

**PURPOSE:** Groups the slot names of a class specified by the CLIPS supplied argument into a multifield variable.

**ARGUMENTS:** A pointer to a data object to hold the resulting multifield.

**OTHER NOTES:** Implementation of the CLIPS function `class-slots`.

### ClassSubclassesCmd

**PURPOSE:** Groups the subclass names of a class specified by the CLIPS supplied argument into a multifield variable.

**OTHER NOTES:** Implementation of the CLIPS function `class-subclasses`.

### ClassSuperclassesCmd

**PURPOSE:** Groups the superclass names of a class specified by the CLIPS supplied argument into a multifield variable.

**ARGUMENTS:** A pointer to a data object to hold the resulting multifield.

**OTHER NOTES:** Implementation of the CLIPS function `class-superclasses`.

### CmdListDefclasses

**PURPOSE:** Lists all the currently defined defclasses.

**OTHER NOTES:** Implementation of the CLIPS function `list-defclasses`.

### CmdUndefclass

**PURPOSE:** Removes a defclass as well as any subclasses and associated instances and message-handlers.

**OTHER NOTES:** Implementation of the CLIPS function `undefclass`.

### DescribeClassCmd

**PURPOSE:** Displays the detailed information about the defclass specified by the CLIPS supplied argument.

**OTHER NOTES:** Implementation of the CLIPS function `describe-class`.
### DoesClassExist

**PURPOSE:** Determines if a class specified by the CLIPS supplied argument exists.

**RETURNS:** A non-zero integer if the class exists, zero otherwise.

**OTHER NOTES:** Implementation of the CLIPS function `class-existp`.

### Embedded Access for Defclasses

**PURPOSE:** The following functions are provided for embedded access and are documented in the *Advanced Programming Guide*: `BrowseClass`, `DeleteDefclass`, `DescribeClass`, `FindDefclass`, `GetClassMessageHandlers`, `GetClassSlots`, `GetClassSubclasses`, `GetClassSuperclasses`, `GetDefclassName`, `GetDefclassPPForm`, `GetNextDefclass`, `GetSlotFacets`, `GetSlotSources`, `IsClassAbstract`, `IsDefclassDeletable` and `ListDefclasses`.

### HasSuperclass

**PURPOSE:** Determines if a class is a subclass of a second class.

**ARGUMENTS:** Pointers to two defclasses.

**RETURNS:** A non-zero integer if the first class is a subclass of the second class, zero otherwise.

**OTHER NOTES** Support routine for superclass and subclass determinant routines in the Class Commands, Generic Function Commands and Generic Function Functions Modules.

### IsClassAbstractCmd

**PURPOSE:** Determines if direct instances of a class specified by the CLIPS supplied argument can be made.

**RETURNS:** The integer zero if the class is abstract, non-zero otherwise.

**OTHER NOTES:** Implementation of the CLIPS function `class-abstractp`.

### IsSubclass

**PURPOSE:** Determines if a class is a subclass of a second class. Both arguments are supplied by CLIPS.
RETURNS: A non-zero integer if the first class is a subclass of the second class, zero otherwise.

OTHER NOTES: Implementation of the CLIPS function `subclassp`.

### IsSuperclass

PURPOSE: Determines if a class is a superclass of a second class. Both arguments are supplied by CLIPS.

RETURNS: A non-zero integer if the first class is a superclass of the second class, zero otherwise.

OTHER NOTES: Implementation of the CLIPS function `superclassp`.

### ObjectsRunTimeInitialize

PURPOSE: Initializes COOL constructs in a run-time image.

ARGUMENTS: 1) Pointer to new class list. 2) Pointer to new definstances list. 3) Pointer to new class hash table.

### PPDefclass

PURPOSE: Displays the pretty-print form of the defclass specified by the CLIPS supplied argument.

OTHER NOTES: Implementation of the CLIPS function `ppdefclass`.

### SetupClasses

PURPOSE: Defines all functions and commands for the defclass construct. Sets up all necessary `load`, `clear`, `save`, `watch`, `constructs-to-c` and `bload/bsave` interfaces.

OTHER NOTES: Initialization differs between standard and run-time configurations.

### SetupObjectSystem

PURPOSE: Initializes all COOL constructs, functions and data structures.

### SlotFacetsCmd

PURPOSE: Groups the facet names of a class slot specified by the CLIPS supplied arguments into a multifield variable.

ARGUMENTS: A pointer to a data object to hold the resulting multifield.
OTHER NOTES: Implementation of the CLIPS function slot-facets.

**SlotSourcesCmd**

PURPOSE: Groups the source class names of a class slot specified by the CLIPS supplied arguments into a multifield variable.

ARGUMENTS: A pointer to a data object to hold the resulting multifield.

OTHER NOTES: Implementation of the CLIPS function slot-sources.

**INTERNAL FUNCTIONS**

**CheckClass**

PURPOSE: Support routine for PPDefclass and DescribeClassCmd which verifies the existence of a class.

ARGUMENTS: 1) The name of the calling function. 2) The name of the class.

RETURNS: A pointer to the class, NULL on errors.

**CheckClassAndSlot**

PURPOSE: Support routine for ClassHasSlot, SlotFacetsCmd and SlotSourcesCmd which parses a class name and a slot name.

ARGUMENTS: 1) The name of the calling function. 2) A buffer for the class pointer.

RETURNS: The symbolic name of the slot, NULL on errors.

**CheckTwoClasses**

PURPOSE: Support routine for IsSuperclass and IsSubclass which parses two class arguments.

ARGUMENTS: 1) The name of the calling function. 2) A buffer for the first class pointer. 3) A buffer for the second class pointer.

RETURNS: A non-zero integer if both classes successfully parsed, zero otherwise.
## ClassInfoFnxArgs

**PURPOSE:** Support routine for `ClassSlotsCmd`, `ClassSuperclassesCmd`, `ClassSubclassesCmd` and `ClassHandlersCmd` which checks the class argument.

**ARGUMENTS:**
1) Name of the calling function.
2) Data object buffer (which is initialized to the symbol FALSE)
3) A buffer for an integer flag indicating if the keyword "inherit" was present in the function call.

**RETURNS:** A pointer to the class, NULL on errors.

## CountSubclasses

**PURPOSE:** Support routine for `GetClassSubclasses` which counts the number of subclasses for a class.

**ARGUMENTS:**
1) A pointer to a class.
2) An integer flag indicating whether to include (one) or exclude (zero) indirectly inherited subclasses.
3) A unique traversal integer identifier to prevent loops when examining the class hierarchy (see the general notes for the Class Commands Module)

**RETURNS:** The number of direct or indirect subclasses (depending on the second argument).

## DisplayHandlersInLinks

**PURPOSE:** Support routine for `DescribeClass` which displays the message-handlers for a list of classes.

**ARGUMENTS:** A list of classes.

**RETURNS:** A non-zero integer if any message-handlers were listed, zero otherwise.

## EvaluateDefaultSlots

**PURPOSE:** Support routine for `ParseDefclass` which evaluates the default value expressions for class slots and converts them to constant expressions.

**ARGUMENTS:** A pointer to the class.

**RETURNS:** A non-zero integer if there are no errors, zero otherwise.
### GetClassName

**PURPOSE:** Support routine for `CmdUndefclass`, `PPDefclass` and `DescribeClassCmd` which parses a class name.

**ARGUMENTS:** The name of the calling function.

**RETURNS:** The name of the class, NULL on errors.

### ParseDefclass

**PURPOSE:** Used by the `load` command to parse a defclass.

**ARGUMENTS:** The logical name of the input source.

**RETURNS:** The integer zero if there are no parsing errors, non-zero otherwise.

### ParseDefclassName

**PURPOSE:** Support routine for `ParseDefclass` which parses a defclass name and optional comment.

**ARGUMENTS:** The logical name of the input source.

**RETURNS:** A pointer to the symbolic name of the new class, NULL on errors.

### ParseSlot

**PURPOSE:** Support routine for `ParseDefclass` which parses slots of a new class.

**ARGUMENTS:** 1) The logical name of the input source. 2) A pointer to the current list of slots.

**RETURNS:** A pointer to the new list of slots, NULL on errors.

### ParseSlotValue

**PURPOSE:** Support routine for `ParseSlot` which parses the value expression for a new slot.

**ARGUMENTS:** 1) The logical name of the input source. 2) A buffer for an error boolean flag.

**RETURNS:** An pointer to an expression.
**ParseSuperclasses**

PURPOSE: Support routine for `ParseDefclass` which parses the direct superclass list of a new class.

ARGUMENTS: 1) The logical name of the input source.
            2) The symbolic name of the new class.

RETURNS: A pointer to a list of classes, NULL on errors.

---

**PrintClassBrowse**

PURPOSE: Support routine for `BrowseClass` which displays the subclasses of a specified class.

ARGUMENTS: 1) A pointer to the root class from which to start the graph.
            2) The depth in the graph from the base class.

---

**PurgeUserClassStuff**

PURPOSE: Used by the `clear` command to remove all currently defined defclasses and their instances.

OTHER NOTES: Defclasses are removed after defmethods but before defgenerics because methods can refer to classes but classes can refer to generic functions. The use of priorities in `AddClearFunction` accomplishes this ordering.

---

**SaveDefclasses**

PURPOSE: Used by the `save` command to write out the pretty-print forms of all the currently defined defclasses.

ARGUMENTS: The logical name of the output destination.

---

**StoreSubclasses**

PURPOSE: Support routine for `GetClassSubclasses` which stores the subclasses of a class in a multifield.

ARGUMENTS: 1) A multifield buffer to store the subclass names.
            2) An index into the multifield buffer indicating where to start the storage of subclass names.
            3) A pointer to a class.
            4) An integer flag indicating whether to include (one) or exclude (zero) indirectly inherited subclasses.
            5) A unique traversal integer identifier to prevent loops when examining the class hierarchy (see the general notes for the Class Commands Module)
RETURNS: The number of direct or indirect subclasses stored in the multifield (depending on the second argument).
The Class Functions Module (classfun.c) handles all the internal manipulations of classes including the construction of class precedence lists from multiple inheritance. For a description of the defclass construct, see the Basic Programming Guide. The defclass construct capability, along with the other features of the CLIPS Object-Oriented Language (COOL), can be removed by using the appropriate compile flag in the setup header file.

**Temporary Instance Slot Link** is an intermediary data structure used to create a template of all the slots which will be present in instances of a class, including slots directly inherited from the class and indirectly inherited from superclasses. FormInstanceTemplate calls MergeSlots for each class in the precedence list to make a list of these temporary slot links. Once the list is complete, the list is converted into the contiguous arrays described in the Class and Instance Commands Modules and then destroyed. The fields of a slot link are: a slot descriptor pointer, an integer index into the inheritance order of the slots (used only when creating the contiguous hash-value map), a link chaining the slots together in order according to increasing hash value of the slot name symbols and a link chaining the slots together in order according to least specific to most specific inheritance.

**Partial Order Link** is an intermediary data structure used to build the class precedence list for a class from the multiple inheritance rules given in the Basic Programming Guide. A partial order for two classes is an assertion that class A must come before or after class B. The multiple inheritance rules are recursively applied to the direct superclasses of a new class to generate a set of partial order links called the partial order table. These partial orders are then topologically sorted according to the algorithm given later in this section to generate the final class precedence list. A partial order table node is comprised of a pointer to a class indicating how many classes (of the ones in the table) must precede that class, and a list of classes which must succeed that class. Specifically, the fields of a partial order link are: a pointer to a class
(the predecessor count is stored with class), a link to the next unrelated partial order and a link to the successor partial orders for this class.

The precedence determination and cycle detection algorithms are adapted from the topological sorting algorithms given in *The Art of Computer Programming - Vol. I (Fundamental Algorithms)* by Donald Knuth.

Each class and its direct superclasses are recursively entered in order into a table of partial orders. A class is only entered once. The order reflects a pre-order depth-first recursive traversal of the classes direct superclass lists, and this order will be followed as closely as possible to preserve the "family" heuristic when constructing the class precedence list.

Attached to each node is a count indicating the number of classes which must precede this class and a list of classes which must succeed this class. These predecessor counts and successor lists indicate the partial orderings given by the rules of multiple inheritance for the classes as given in the *Basic Programming Guide*.

**Rules of Multiple Inheritance:**

1. A class must precede all its superclasses.
2. A class determines the precedence of its direct superclasses.

For example, the following class definitions:

(defclass A (is-a USER))
(defclass B (is-a USER))
(defclass C (is-a A B))

would give the following partial orders:

<table>
<thead>
<tr>
<th>Partial Order</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>C &lt; A</td>
<td>Rule 1</td>
</tr>
<tr>
<td>C &lt; B</td>
<td>Rule 1</td>
</tr>
<tr>
<td>A &lt; B</td>
<td>Rule 2</td>
</tr>
<tr>
<td>B &lt; USER</td>
<td>Rule 1</td>
</tr>
<tr>
<td>A &lt; USER</td>
<td>Rule 1</td>
</tr>
<tr>
<td>USER &lt; OBJECT</td>
<td>Rule 1</td>
</tr>
</tbody>
</table>

Entering these partial orders into a table using partial order links would yield the following (note that the predecessor count is actually stored in the defclass data structure, not the partial order link):
To generate a precedence list for the class C, pick the first class (scanning from left to right) with a predecessor count of 0, append it to the precedence list, and decrement the counts of all its successors. Continue scanning for a zero from where the last scan left off. If there are no classes left with a predecessor count of zero, then there is no solution. The function \textit{PrintPartialOrderLoop} implements a straightforward algorithm to print out a cyclical dependency in the partial orders when an error is detected.

If the algorithm were not concerned about preserving the "family" heuristic (i.e. trying to match pre-order depth-first traversal as closely as possible), neither the order in which the classes were entered into the table nor the order in which the table was scanned for classes with predecessor counts of zero would matter. Picking only classes which have a predecessor count of zero guarantees a solution (if it exists) that satisfies the two multiple inheritance rules. The modifications to Knuth's algorithm allow the additional heuristic to be observed.

The following diagrams show the partial order table after each class is entered onto the precedence list:

**Precedence List so far: C**

**Precedence List so far: C A**
GLOBAL VARIABLES

<table>
<thead>
<tr>
<th>ClassList</th>
</tr>
</thead>
</table>

PURPOSE: A pointer to the first node in the list of all currently defined defclasses.

OTHER NOTES: The primitive CLIPS type classes come first in this list according to the integer codes in constant.h. For example, the codes for FLOAT and INTEGER are 0 and 1, thus those classes come first and second on the list respectively. The purpose of this ordering is to make the binary save files identical for generic functions whether or not COOL is installed (see the Generic Functions Binary Load/Save Interface Module for more details).

The order of primitive type classes in PrimitiveClassMap is a mirror image of this mapping.
**ClassTable**

**PURPOSE:** An array of class lists where each class in a particular list has the same hash value. This data structure enables fast lookups of classes by name. The class name is hashed to generate a hash value, and then compared to the names of all the classes in the chain at the hash value index of the class table.

**CLASS_TABLE_HASH_SIZE**

**PURPOSE:** The number of chains in the class lookup table. A chain is a list of classes where the name of each class generates the same hash value according to the formula in HashClass.

**OTHER NOTES:** Implemented as a preprocessor constant in classfun.h.

**OBJECT_CLASS_STRING**

**PURPOSE:** The name of the root class of all classes in COOL; this class has no superclasses.

**OTHER NOTES:** Implemented as a preprocessor constant in classfun.h.

**PrimitiveClassMap**

**PURPOSE:** An array of class pointers for the basic CLIPS primitive types where the position of the array corresponds to the integer code of the type found in constant.h. For example, the pointer to the EXTERNAL-ADDRESS class is found in PrimitiveClassMap[5] since the code for EXTERNAL-ADDRESS is 5.

**OTHER NOTES:** The order of primitive type classes in ClassList is a mirror image of this mapping.

**USER_CLASS_STRING**

**PURPOSE:** The name of the base class of user-defined classes; this class has all the predefined message-handlers attached to it.

**OTHER NOTES:** Implemented as a preprocessor constant in classfun.h.
INTERNAL VARIABLES

BIG_PRIME

PURPOSE: Large prime number used in the calculations of widely distributed hash values for classes.

OTHER NOTES: Implemented as a preprocessor constant.

ClassListBottom

PURPOSE: A pointer to the last node in the list of all currently defined defclasses.

Instance Template Codes

PURPOSE: Integer codes used by the function MergeSlots to indicate that a list of slots are being inherited by a new class from a direct (DIRECT) or indirect (INDIRECT) superclass.

OTHER NOTES: Used in connection with the no-inherit facet for a slot.

GLOBAL FUNCTIONS

AddClass

PURPOSE: Support routine for ParseDefclass in the Class Commands Module which inserts a new class into the global list and hash table and deletes an old definition, if necessary.

ARGUMENTS: A pointer to a new class.

OTHER NOTES: For a class redefinition, old message-handlers which do not conflict with new slot-accessors are reattached to the new class.

ClassExistError

PURPOSE: Prints out an error message when a class cannot be found for various functions.

ARGUMENTS: 1) The name of the calling function.
2) The name of the non-existent class.
**ClearDefclasses**

**PURPOSE:** Deletes all user-defined classes and message-handlers.

**ARGUMENTS:** An integer code indicating whether to ignore (zero) or delete (non-zero) user-defined message-handlers attached to system classes.

**RETURNS:** A non-zero integer if all user-defined classes and message-handlers are deleted, zero otherwise.

**OTHER NOTES:** Classes which is in use will not be deleted.

**DeleteClassLinks**

**PURPOSE:** Support routine for **ParseDefclass** in the Class Commands Module which deallocates a temporary set of links forming a list of classes.

**ARGUMENTS:** A pointer to a class link node.

**DeleteClassUAG**

**PURPOSE:** Removes a class and all its subclasses.

**ARGUMENTS:** A pointer to a class.

**RETURNS:** A non-zero integer if the class and its subclasses, including associated message-handlers, are deleted, zero otherwise.

**OTHER NOTES:** A class which is in use will not be deleted.

**DeleteSlots**

**PURPOSE:** Support routine for **ParseDefclass** in the Class Command Module which deletes a list of slots.

**ARGUMENTS:** The list of slots.

**OTHER NOTES:** The "shared value" data object field of the slots is temporarily used as a "next" pointer to chain together slots in the list.

**FindClassSlot**

**PURPOSE:** Determines the address of a specified class slot.

**ARGUMENTS:**
1) A pointer to a class.
2) A symbolic slot name.
**FindDefclassBySymbol**

**PURPOSE:** Determines the address of a specified class.

**ARGUMENTS:** A pointer to a symbol.

**RETURNS:** A pointer to a class.

**FindPrecedenceList**

**PURPOSE:** Support routine for `ParseDefclass` in the Class Commands Module which determines the class precedence list for a new class using the multiple inheritance rules explained in Section 9.3.1.1 of the *Basic Programming Guide*. Using these rules, partial orders between the superclasses of the new class are generated. A topological sort is used to establish a linear ordering of all the superclasses from the partial orders; this ordering is referred to as the class precedence list. The algorithm is explained in detail in the general notes for the Class Functions Module.

**ARGUMENTS:**
1) A pointer to the old class for which the precedence list is being redefined (NULL if this is a new class).
2) A list of direct superclasses.

**RETURNS:** A list of classes forming the precedence list, or NULL on errors.

**InitializeClasses**

**PURPOSE:** Allocates and initializes class hash table and creates system classes.

**InsertSlot**

**PURPOSE:** Support routine for `ParseDefclass` in the Class Commands Module which inserts a new slot into the list of slots for a new class and verifies that it is not a duplicate.

**ARGUMENTS:**
1) The top of the slot list.
2) A pointer to a slot.

**RETURNS:** The top of the slot list.

**OTHER NOTES:** The "shared value" data object field of the slots is temporarily used as a "next" pointer to chain together slots in the list.
InstancesPurge

PURPOSE: Removes all instances of user-defined classes.

IsClassBeingUsed

PURPOSE: Recursively checks to see if a class or any of its subclasses are in use.

ARGUMENTS: A pointer to a class.

RETURNS: A non-zero integer if the class or any of its subclasses is busy, zero otherwise.

IsSystemClassName

PURPOSE: Support routine for ParseDefclass in the Class Commands Module which determines if a name matches that of one of the predefined system classes.

ARGUMENTS: The symbolic name of a class.

RETURNS: A non-zero integer if the name matches a system class name, zero otherwise.

NewClass

PURPOSE: Allocates and initializes a new class.

RETURNS: A pointer to an initialized class.

OTHER NOTES: A traversal id map of size TRAVERSAL_BYTES (see the Instance-Set Queries Module) is allocated for the class.

NewSlot

PURPOSE: Allocates and initializes a new slot.

ARGUMENTS: The symbolic slot name.

RETURNS: A pointer to a new slot.

OTHER NOTES: Creates two new symbols called "get-"<slot-name> and "put-"<slot-name> and initializes the "related symbol" fields of these symbols to point to the original slot name. This is to help with fast lookup of implicit slot accessor message-handlers during a message dispatch (see the Message Functions Module).
**ObjectSystemPurge**

**PURPOSE:** Removes all definstances and user-defined classes and associated instances.

**PackSlots**

**PURPOSE:** Support routine for `ParseDefclass` in the Class Commands Module which packs a list of slots into a contiguous array for easy reference.

**ARGUMENTS:**
1) A pointer to the class.
2) A list of slots.

**PutClassInTable**

**PURPOSE:** Inserts a class into the class hash table.

**ARGUMENTS:** A pointer to a class.

**ReinitializeClasses**

**PURPOSE:** Initializes (but does not allocate) class hash table and creates system classes.

**SetClassList**

**PURPOSE:** Initializes the global variables `ClassList` and `ClassListBottom` to point to the top and bottom respectively of the given list of classes.

**ARGUMENTS:** A pointer to the top of a list of classes.

**OTHER NOTES:** This function is used only in a run-time version of CLIPS.

**INTERNAL FUNCTIONS**

**AddSystemClass**

**PURPOSE:** Support routine for `CreateSystemClasses` which allocates and initializes a new system class and inserts into the class hash table.

**ARGUMENTS:**
1) The name of the class.
2) The CLIPS type integer code found in constant.h which corresponds to this class (any value if this does not apply, e.g. the USER class).
3) An integer flag indicating if the new class corresponds to a
CLIPS primitive type (1) or not (0).
4) A pointer to the parent class, or NULL if there is none.

RETURNS: A pointer to a class.

OTHER NOTES: This function assumes simple single inheritance for the system classes and builds class precedence lists accordingly.

---

**AddToClassList**

PURPOSE: Support routine for CreateSystemClasses which adds a class to the end of the global class list.

ARGUMENTS: A pointer to the class.

---

**BuildPartialOrders**

PURPOSE: Support routine for FindPrecedenceList which builds a table of the partial orders between the superclasses of a new class based on the multiple inheritance rules and "family" heuristic specified in section 9.3.1.1 of the Basic Programming Guide. See the general notes for the Class Functions Module for more details.

ARGUMENTS: 1) The partial order table, which is a series of partial order links (see the general notes). 2) The list of superclasses.

RETURNS: The partial order table.

OTHER NOTES: If a class is being redefined, the Multiple Inheritance Rule 1 partial orders between the class and its direct superclasses will have already been recorded in the table prior to the calling of this function. Otherwise, the partial order table will be empty. This is done to prevent cyclical dependencies.

---

**BuildSubclassLinks**

PURPOSE: Support routine for AddClass which attaches subclass links from each superclass of a class to itself.

ARGUMENTS: A pointer to a class.

---

**CopyClassLinks**

PURPOSE: Support routine for AddSystemClass which copies a list of classes to aid in creating the class precedence list for a new system class.
ARGUMENTS: A list of classes.

**CreateSystemClasses**

PURPOSE: Creates the predefined system classes and adds them to the global class list and hash table.

OTHER NOTES: The order in which the system classes are defined is important. See the notes on the Class Functions Module global variables `ClassList` and `PrimitiveClassMap`.

**DeleteClass**

PURPOSE: Support routine for `ObjectSystemPurge`, `ClearDefclasses` and `DeleteClassUAG` which deinstalls (see `InstallClass`) and deallocates a class and its message-handlers.

ARGUMENTS: A pointer to a class.

RETURNS: A non-zero integer if the class was successfully deleted, zero otherwise.

OTHER NOTES: This routine will fail if there are any outstanding references to the class (e.g. instances or subclasses still exist).

**DeleteSublink**

PURPOSE: Support routine for `DeleteClass` which removes the subclass link to a class from one of its superclasses.

ARGUMENTS: 1) A pointer to the superclass. 2) A pointer to the subclass.

**FormInstanceTemplate**

PURPOSE: Support routine for `AddClass` which creates a template of all the slots which will be present in instances of a class, including slots directly inherited from the class and indirectly inherited from superclasses. This template is a contiguous array of class slot pointers sorted by inheritance from least specific to most specific (slots from the same class are in the order they appeared in the defclass). This ordering is used when listing slot information about the class. This data structure is explained in detail in the Class and Instance Commands Modules. All direct instances of a class share the same instance template; each instance only needs to have its own array of slot values.
ARGUMENTS: A pointer to the class.

OTHER NOTES: This routine also creates a map of integer indices into the instance template which gives the order according to the hash values of the symbolic names of the slots. Thus, slots for an instance can be easily found by performing a binary search on the symbolic hash value of the slot name.

Temporary instance slot links are used as an intermediary data structure.

---

**HashClass**

PURPOSE: Generates an index into the class hash table for a given class.

ARGUMENTS: The symbolic name of a class.

RETURNS: The hash table index for the class.

OTHER NOTES: The hash table index is derived from the symbol hash table index (see the Symbol Manager Module).

---

**InstallClass**

PURPOSE: Support routine for **AddClass** and **DeleteClass** which increments or decrements the "in use" counts of all atoms (e.g. symbols) and construct references (e.g. deffunction calls) found in the definitions of a class and its associated message-handlers. This insures that all of these items persist at least as long as the class definition does.

ARGUMENTS: 
1) A pointer to a class.
2) An integer code indicating whether to increment (1) or decrement (0) "in use" counts.

OTHER NOTES: Only deinstallation of message-handlers is performed by this routine; installation is done by **ParseDefmessageHandler** in the Message Commands Module.

---

**MergeSlots**

PURPOSE: Support routine for **FormInstanceTemplate** which appends slots from a class to a temporary list of slots inherited from more specific classes. Only slots which have not already been specified will be added, and slots with **no-inherit** facets will only be added from the direct parent class, not indirect superclasses.
ARGUMENTS: 1) The current list of temporary slot links.  
2) A pointer to a class with the new slots.  
3) Buffer for the number of slots in the list.  
4) An integer flag indicating if the new slots to be added are from the direct parent class (1) or not (0).  
5) Buffer for holding the top of the list according to most specific order by inheritance.

RETURNS: The new list of temporary instance slot links.

OTHER NOTES: The temporary link data structure uses a "next" field for indicating the symbolic hash value order and an "inherit" field for indicating the most specific inheritance order.

PrintPartialOrderLoop

PURPOSE: Support routine for FindPrecedenceList which prints a dependency loop in the partial orders when a precedence list cannot be found which satisfies the multiple inheritance rules. Details are given in the general notes.

ARGUMENTS: The table of partial orders.

OTHER NOTES: There may be more than one dependency loop between the partial orders, but this routine will only print the first one it finds.

RecordPartialOrder

PURPOSE: Support routine for FindPrecedenceList and BuildPartialOrders which enters a partial order between two classes into the partial order table, e.g. class A has precedence over class B. According to the topological sorting algorithm given in the Class Function Module's general notes, the successor class is entered on the predecessor class's successor list, and the successor class's predecessor count is incremented.

ARGUMENTS: 1) A partial order link node containing a pointer to the predecessor class.  
2) The successor class.

ResetCompositeSlots

PURPOSE: Support routine for AddClass which gets facet values for composite slots in a new class from its superclasses. See the Basic Programming Guide for details on the composite slot facet.
ARGUMENTS: A pointer to a class.

OTHER NOTES: Since all superclasses are completely defined before a new class based on them is created, this routine need only examine the immediately next most specific class in the class precedence list for extra facet values. Even if the superclass slot is also composite, the other facets have already been filtered down from the more general superclasses. However, if the superclass slot has a no-inherit facet, the next most specific class must be examined.
The Instance Commands Module (inscom.c) manages the parsing and general interface aspects for instances of user-defined classes. For a description of how to manipulate instances, see the Basic Programming Guide.

The internal data structure of an instance consists primarily of: a symbolic name, a pointer to the class (see the Class Commands Module for details on the defclass construct) and an array of slot values corresponding one to one with the instance template array in the class. A busy count for each instance reflects how many outstanding pointer references there are to an instance. This busy count must be zero if it is safe to deallocate the instance. Note that the instance may appear to be deleted while the busy count is greater than zero, for it might be on the garbage collection list (see the notes in the Instance Functions Module).

Other fields in the instance structure include: a hash value indicating the position of the instance in the hash table; the evaluation depth at which the instance was created (see the Evaluation Module), which is used in determining when it is safe to garbage collect an instance; a series of bitfields indicating such things as whether the instance is on the garbage list; and links connecting the instance to the global instance list and hash table.

The bitfields are stored in a single integer and indicate the following information about an instance: whether all the atoms within the instance have had their busy counts incremented (i.e. whether the instance has been installed), whether the instance has been functionally deleted (i.e. the instance on the garbage list) and whether the instance is in the process of being initialized by make-instance or initialize-instance.
The links which place an instance in the global instance list (InstanceList), the instance hash table (InstanceTable) and the list of instances for the parent class are implemented with direct instance pointers in the instance.

Each instance has an array of instance slots that correspond one to one with the template of slot descriptor pointers in the instance's class. In this way, slot information which is common to all instances of a class, such as the slot name and facets, is only stored once. The mapping is one to one so that an instance slot may be accessed by looking it up by name in the class template and then using the same index to reference the instance slot array. The instance slots contain information about slots which are specific to each instance.

An instance slot contains the following fields: a pointer to the slot descriptor giving the name and facets; a data object buffer holding the slot value, if the slot is local; an expression for the initial value of a slot used during make-instance and initialize-instance; and a pointer to the data object buffer holding the slot value. This last field will point to the local value field if the slot is not a shared slot. Otherwise, it will point to the shared value field in the slot descriptor of the class. This field is used to access the slot value indirectly COOL routines to avoid repetitive checks as to whether the slot is shared or not.
INTERNAL VARIABLES

**ALL_QUALIFIER**

**PURPOSE:** Keyword which tells `CmdListInstances` to list indirect instances of a class.

**OTHER NOTES:** Implemented as a preprocessor constant.

**CLASS_RLN**

**PURPOSE:** Keyword in instance creation routines `ParseInitializeInstance` and `ParseSimpleInstance` indicating that a class name follows.

**OTHER NOTES:** Implemented as a preprocessor constant.

**ObjectParseToken**

**PURPOSE:** An intermediary variable used for scanned tokens by the COOL parsing routines during a *load*.

GLOBAL FUNCTIONS

**CmdListInstances**

**PURPOSE:** Lists instances of a class.

**OTHER NOTES:** Implementation of the CLIPS function `instances`.

**DeleteInstance**

**PURPOSE:** Deletes the active instance, i.e. the instance which is the object of the current message.

**OTHER NOTES:** Implementation of the CLIPS function `delete-instance`.

**DestroyAllInstances**

**PURPOSE:** Sends `delete` messages to all instances.

**DoesInstanceExist**

**PURPOSE:** Determines if the instance specified by the instance name or address in the CLIPS supplied argument exists.

**RETURNS:** A non-zero integer if the instance exists, zero otherwise.
OTHER NOTES: Implementation of the CLIPS function `instance-existp`.

### DoesSlotExist

**PURPOSE:** Determines if a slot of a particular instance, specified by CLIPS supplied arguments, exists.

**RETURNS:** A non-zero integer if the slot exists, zero otherwise.

OTHER NOTES: Implementation of the CLIPS function `slot-existp`.

### Embedded Access for Instances

**PURPOSE:** The following functions are provided for embedded access and are documented in the *Advanced Programming Guide*: `CLIPSDeleteInstance`, `CLIPSGetSlot`, `CLIPSMakeInstance`, `CLIPSPutSlot`, `CLIPSTestSlot`, `CLIPSUnmakeInstance`, `CreateRawInstance`, `FindInstance`, `GetInstanceName`, `GetInstanceClass`, `GetInstancePPForm`, `GetNextInstance`, `GetNextInstanceInClass`, `ListInstances`, `LoadInstances`, `SaveInstances` and `ValidInstanceAddress`.

OTHER NOTES: There are additional embedded access functions for instances in the Instances Functions Module.

### GetInstanceAddressCmd

**PURPOSE:** Determines the address of an instance specified by a CLIPS supplied argument.

**ARGUMENTS:** A data object buffer to hold the instance address.

OTHER NOTES: Implementation of the CLIPS function `instance-address`.

### GetInstanceClassCmd

**PURPOSE:** Determines the class of an instance specified by a CLIPS supplied argument.

**ARGUMENTS:** A data object buffer to hold the class name.

OTHER NOTES: Implementation of the CLIPS functions `class` and `type` (see the Generic Function Command Module).
**GetInstanceNameCmd**

**PURPOSE:** Determines the name of an instance specified by a CLIPS supplied argument.

**ARGUMENTS:** A data object buffer to hold the instance name.

**OTHER NOTES:** Implementation of the CLIPS function `instance-name`.

**InstanceNameToSymbol**

**PURPOSE:** Converts an instance name specified by a CLIPS supplied argument to a symbol.

**ARGUMENTS:** A data object buffer for holding the symbol.

**OTHER NOTES:** Implementation of the CLIPS function `instance-name-to-symbol`.

**IsInstance**

**PURPOSE:** Determines if the CLIPS supplied argument is an instance name or address.

**RETURNS:** A non-zero integer if the argument is an instance, zero otherwise.

**OTHER NOTES:** Implementation of the CLIPS function `instancep`.

**IsInstanceAddress**

**PURPOSE:** Determines if the CLIPS supplied argument is an instance address.

**RETURNS:** A non-zero integer if the argument is an instance address, zero otherwise.

**OTHER NOTES:** Implementation of the CLIPS function `instance-addressp`.

**IsInstanceName**

**PURPOSE:** Determines if the CLIPS supplied argument is an instance name.

**RETURNS:** A non-zero integer if the argument is an instance name, zero otherwise.

**OTHER NOTES:** Implementation of the CLIPS function `instance-namep`. 
<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>Arguments</th>
<th>Other Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IsSlotBound</strong></td>
<td>Determines if the slot of an instance, specified by CLIPS supplied arguments, has a bound value.</td>
<td></td>
<td>Implementation of the CLIPS function <code>slot-boundp</code>.</td>
</tr>
<tr>
<td><strong>IsSlotInitable</strong></td>
<td>Determines if the slot of an instance, specified by CLIPS supplied arguments, can be initialized.</td>
<td></td>
<td>Implementation of the CLIPS function <code>slot-initablep</code>.</td>
</tr>
<tr>
<td><strong>IsSlotWritable</strong></td>
<td>Determines if the slot of an instance, specified by CLIPS supplied arguments, can be written.</td>
<td></td>
<td>Implementation of the CLIPS function <code>slot-writablep</code>.</td>
</tr>
<tr>
<td><strong>LoadInstancesCommand</strong></td>
<td>Loads instances from a file.</td>
<td></td>
<td>Implementation of the CLIPS function <code>load-instances</code>.</td>
</tr>
<tr>
<td><strong>MultifieldSlotDelete</strong></td>
<td>Deletes fields from a multifield slot value of an instance.</td>
<td>A data object buffer to hold the symbol TRUE or FALSE depending on the success of the deletion.</td>
<td>Implementation of the CLIPS function <code>mv-slot-delete</code>.</td>
</tr>
<tr>
<td><strong>MultifieldSlotInsert</strong></td>
<td>Inserts fields into a multifield slot value of an instance.</td>
<td>A data object buffer to hold the symbol TRUE or FALSE depending on the success of the insertion.</td>
<td></td>
</tr>
</tbody>
</table>
OTHER NOTES: Implementation of the CLIPS function `mv-slot-insert`.

**MultifieldSlotReplace**

**PURPOSE:** Replaces fields in a multifield slot value of an instance.

**ARGUMENTS:** A data object buffer to hold the symbol TRUE or FALSE depending on the success of the replacement.

OTHER NOTES: Implementation of the CLIPS function `mv-slot-replace`.

**ParseInitializelInstance**

**PURPOSE:** Parses `initialize-instance` and `make-instance` function calls into a series of expressions that can later be evaluated by `InitializeInstance` or `MakeInstance` in the Instance Functions Module.

**ARGUMENTS:**
1) An expression node containing the `initialize-instance` or `make-instance` function call.
2) The logical name of the input source.

**RETURNS:** The top of series of expressions representing the `initialize-instance` or `make-instance` function call, or NULL on errors.

OTHER NOTES: This special function parser is required because these two functions do not follow the standard format of CLIPS functions, e.g. the slot-overrides would look like function calls to the standard CLIPS function parser.

`initialize-instance <instance> <slot-override>*`) is parsed to the following:

```
| Instance Expression | NEXT-ARG | Slot name Expression | NEXT-ARG | Symbol | TRUE |
```

`make-instance <instance-name> of <class> <slot-override>*`) is parsed to the following:
**PPInstance**

PURPOSE: Displays the pretty-print form of an instance.

OTHER NOTES: Implementation of the CLIPS function `ppinstance`.

**SaveInstancesCommand**

PURPOSE: Saves instances to a file.

OTHER NOTES: Implementation of the CLIPS function `save-instances`.

**SetupInstances**

PURPOSE: Support routine for `SetupObjectSystem` in the Class Commands Module which defines all functions and commands for instances. Sets up all necessary watch and garbage collection interfaces.

OTHER NOTES: Initialization differs between standard and run-time configurations.

**SymbolToInstanceName**

PURPOSE: Converts a symbol specified by a CLIPS supplied argument to an instance name.

ARGUMENTS: A data object buffer for holding the instance name.

OTHER NOTES: Implementation of the CLIPS function `symbol-to-instance-name`.

**UnmakeInstance**

PURPOSE: Sends a `delete` message to the instance specified by the CLIPS supplied argument.

OTHER NOTES: Implementation of the CLIPS function `unmake-instance`. 
INTERNAL FUNCTIONS

### CheckInstanceAndSlot

**PURPOSE:** Support routine for `DoesSlotExist`, `IsSlotBound`, `IsSlotWritable` and `IsSlotInitable` which determines the address of a slot of an instance, both given by CLIPS supplied arguments.

**ARGUMENTS:**
1) The name of the calling function.
2) A buffer for holding the slot address (will contain NULL if the slot does not exist).
3) An integer flag indicating whether to signal an error (0) if the slot does not exists or not (1).

**RETURNS:** The address of the instance, NULL on errors.

### CheckMultifieldSlotInstance

**PURPOSE:** Support routine for `MultifieldSlotDelete`, `MultifieldSlotInsert` and `MultifieldSlotReplace` which checks the number of arguments and determines the address of the instance specified by the CLIPS supplied argument.

**ARGUMENTS:**
1) Name of the calling function.
2) Integer code representing a restriction on the number of arguments (EXACTLY, AT_LEAST, NO_MORE_THAN, etc.)
3) The expected number of arguments.

**RETURNS:** The address of the instance, NULL on errors.

### FindISlotByName

**PURPOSE:** Support routine for `CLIPSTestSlot`, `CLIPSGetSlot` and `CLIPSPutSlot` which determines the address of a named slot.

**ARGUMENTS:**
1) The address of an instance.
2) The name of a slot.

**RETURNS:** The address of the instance slot, NULL on errors.

### ParseSimpleInstance

**PURPOSE:** Support routine for `LoadInstances` and `CLIPSMakeInstance` which parses a simple `make-instance` call.
ARGUMENTS: 1) An expression node containing the `make-instance` function call.
2) The logical name of the input source.

RETURNS: The top of series of expressions representing the `make-instance` function call, or NULL on errors.

OTHER NOTES: This routine is similar in functionality to `ParseInitializeInstance`, except that it is always available (even in a run-time or binary load only configuration). It is more constraining than `ParseInitializeInstance` in that the instance name, class name and the slot names and slot values in slot-overrides must all be constant expressions.

### ParseSlotOverrides

PURPOSE: Support routine for `ParseInitializeInstance` which parses slot-overrides in an `initialize-instance` or `make-instance` function call.

ARGUMENTS: 1) The logical name of the input source.
2) An integer buffer for holding a an error code: non-zero on errors, zero otherwise.

RETURNS: A series of expressions representing the slot-overrides.

### PrintInstance

PURPOSE: Support routine for `PPInstance` and `GetInstancePPForm` which prints the name, class and slot values of an instance.

ARGUMENTS: 1) Logical name of the output destination.
2) Address of the instance.
3) String to print between slots.

### TabulateInstances

PURPOSE: Support routine for `ListInstances` which lists and counts all the instances of a class.

ARGUMENTS: 1) A unique traversal integer identifier to prevent loops when examining the class hierarchy (see the general notes for the Class Commands Module)
2) The logical name of the output destination.
3) A pointer to a class.
4) An integer flag indicating whether to print indirect instances of the class (1) or not (0).
RETURNS: The number of instances listed.
The Instance Functions Module (insfun.c) deals with the internal details of creating, accessing and deleting instances.

When an instance is deleted but its busy count is still greater than zero or its creation evaluation depth is less than the current one, it is removed from its class's instance list, the global instance list and hash table, and it is placed in a special delayed garbage collection queue. An intermediary data structure is used to build this garbage list:

```
Instance Garbage Instance (Instance Pointer)
           |                   |
           |                   |
           |                   |
           |                   |
Next Link (Instance Garbage Pointer)
```

The first field is a pointer to the deleted instance, and the second is a pointer to the next node in the garbage list. Whenever an instance is deleted, regardless of whether it is busy or not, all of the internal data of the instance, such as the slot values, are immediately deallocated. However, the external links of the instance are left intact so that any outstanding internal references to the instance can still follow these links without getting an unexpected pointer violation. CLIPS functions available to the user will not be able to access this address anymore, even if the address was bound to a CLIPS variable, because CLIPS recognizes that the instance no longer effectively exists and treats references to it accordingly. This is accomplished through the use of the "garbage" bitfield in the instance (see the notes in the Instance Commands Module). At a later time, when garbage collection is performed and the instance is no longer busy, the actual instance data structure will be deallocated. The Utility and Evaluation Modules provide more details on CLIPS garbage collection.

**GLOBAL VARIABLES**

**ChangesToInstances**

PURPOSE: Internal flag used by the functions `GetInstancesChanged` and `SetInstancesChanged`, which are documented in the Advanced Programming Guide.

**InstanceList**

PURPOSE: A pointer to the top of the list of currently defined instances.

**INSTANCE_TABLE_HASH_SIZE**

PURPOSE: The number of chains in the instance lookup table. A chain is a list of instances where the name of each instance generates the same hash value according to the formula in `HashInstance`.
**OTHER NOTES:** Implemented as a preprocessor constant in insfun.h.

---

**MaintainGarbageInstances**

**PURPOSE:** When this global integer flag is non-zero, instances which are on the garbage collection list cannot be deleted and any newly deleted instances go onto the garbage collection list regardless of whether they are busy or not. This flag is used as a convenient mechanism by various instance manipulation functions to insure that instance addresses remain valid during a certain interval.

---

**Multifield Slot Function Codes**

**PURPOSE:** Integer codes used by CheckMultifieldSlotModify to determine which function called it. The codes are: DELETE, INSERT and REPLACE.

---

**OverrideSlotProtection**

**PURPOSE:** When this integer flag is non-zero, slot protection such as read-only and initialize-only facets are ignored during slot writes. This flag is used by the LoadInstances function in the Instance Commands Module to insure that instances can be reloaded from a file exactly as they were saved.

---

**Slot Value Expression Evaluation Codes**

**PURPOSE:** Integer codes used by EvaluateAndStoreInDataObject to indicate the result of evaluating an expression for a slot value: MULTI_CLEAR for a NIL value and MULTI_SET for a non-NIL value.

---

**Slot Value Set Codes**

**PURPOSE:** Integer codes used by PutSlotValue to indicate the result of setting a slot: SLOT_EMPTY for clearing the slot, SLOT_FILLED for setting the slot and SLOT_ERROR on errors.

---

**OTHER NOTES:** Implemented as preprocessor constants in insfun.h.
WatchInstances

PURPOSE: An integer flag indicating whether or not to print out trace information whenever an instance is created or deleted. This flag is used by the watch command.

WatchSlots

PURPOSE: An integer flag indicating whether or not to print out trace information whenever a slot changes value. This flag is used by the watch command.

WithinInit

PURPOSE: This integer flag is non-zero when an instance is being initialized. This lets other COOL routines know when it is permissible to write to slots which are protected by the initialize-only facet.

INTERNAL VARIABLES

BIG_PRIME

PURPOSE: Large prime number used in the calculations of widely distributed hash values for instances.

OTHER NOTES: Implemented as a preprocessor constant.

CurrentInstance

PURPOSE: A pointer to the instance which is currently being created.

InstanceGarbageList

PURPOSE: A pointer to the top of the list of instances which are functionally deleted but still remain to be garbage collected. Instances will remain on this list as long as their busy count is non-zero; this to insure that there are no dangling pointers.

InstanceListBottom

PURPOSE: A pointer to the bottom of the list of currently defined instances.
InstanceTable

PURPOSE: An array of instance lists where each instance in a particular list has the same hash value. This data structure enables fast lookups of instances by name. The instance name is hashed to generate a hash value and then compared to the names of all the instances in the chain at the hash value index of the instance table.

GLOBAL FUNCTIONS

BuildInstance

PURPOSE: Support routine for CreateRawInstance in the Instance Commands Module and MakeInstance which creates an uninitialized instance and inserts it into the class's instance list and the global instance list and hash table.

ARGUMENTS: 1) Symbolic name of the new instance. 2) Symbolic name of a class.

RETURNS: The address of the new instance, NULL on errors.

OTHER NOTES: If an instance of the specified name already exists, it is sent a delete message. If the deletion fails, the new creation is aborted.

CheckMultifieldSlotModify

PURPOSE: Support routine for the functions MultifieldSlotDelete, MultifieldSlotInsert and MultifieldSlotReplace in the Instance Commands Module and HandlerDeleteSlot, HandlerInsertSlot and HandlerDeleteSlot in the Message-Handler Commands Module which gets the slot address, field range indices and new field values (if any) for these functions.

ARGUMENTS: 1) A code indicating the type of operation (see Multifield Slot Function Codes): INSERT: Requires one index REPLACE: Requires two indices DELETE: Requires two indices 2) Name of the calling function. 3) Pointer to the instance being modified. 4) Argument expressions for the calling function. 5) Integer buffer for the range start index. 6) Integer buffer for the range end index (can be NULL if argument #1 is INSERT).
7) Data object buffer for the new value to be inserted or used as a replacement (can be NULL if argument #1 is DELETE).

RETURNS: The address of the instance slot to modify, NULL on errors.

OTHER NOTES: A multifield value is allocated and added to the ephemeral segment list if more than one new field value is specified.

### CleanupInstances

**PURPOSE:** This function is called by the CLIPS garbage collector, PeriodicCleanup in the Utility Module, to deallocate garbage collected instances which are not busy and which were created at an evaluation depth greater than the current one.

### CoreInitializeInstance

**PURPOSE:** Support routine for InitializeInstance and MakeInstance which performs the following steps to initialize an instance:

1) Get all default slot value expressions from the class definition.
2) Replace default slot value expressions with slot-overrides as appropriate.
3) Evaluate slot-overrides with put messages.
4) Evaluate remaining default slot values via init message.

**ARGUMENTS:**
1) A pointer to the instance.
2) A series of slot-override expressions.

**RETURNS:** A non-zero integer if the instance is successfully initialized, zero otherwise.

**OTHER NOTES:** This function does not need to be global; it will be made an internal function in the next release of CLIPS.

### DecrementInstanceDepth

**PURPOSE:** Support routine for PropogateReturnValue in the Evaluation Module which decrements the evaluation depth of an instance.

**ARGUMENTS:** A pointer to an instance.

### Embedded Access for Instances

**PURPOSE:** The following functions are provided for embedded access and are documented in the Advanced Programming Guide: CLIPS Architecture Manual
**DecrementInstanceCount, GetInstancesChanged, IncrementInstanceCount and SetInstancesChanged.**

**OTHER NOTES:** There are additional embedded access functions for instances in the Instances Commands Module.

---

### EvaluateAndStoreInDataObject

**PURPOSE:** Support routine for `EvaluateDefaultSlots` in the Class Commands Module, `HandlerPutSlot` in the Message-Handler Commands Module, `EvaluateInstanceSlots` and `CheckMultifieldSlotModify` which evaluates a series of expressions and stores the result in a data object.

**ARGUMENTS:**
1) An integer code indicating whether to store the result in an atomic data object (0) or a multifield data object (1), if the expression is atomic (i.e. the "next argument" pointer is NULL).
2) The series of expressions to evaluate.
3) A data object buffer to hold the result.

**RETURNS:** The integer zero if there are any errors while evaluating the expressions, `MULTI_CLEAR` (1) if the expression list is NULL, or `MULTI_SET` (2) otherwise (see Slot Value Expression Evaluation Codes).

---

### EvaluateInstanceSlots

**PURPOSE:** Directly evaluates class default slot expressions for slot values which were not specified by slot-overrides (i.e. the "override" flag was not set for the slot; see the general notes in the Instance Commands Module) in the `make-instance` or `initialize-instance` call.

**ARGUMENTS:** A data object buffer which holds the instance address if there were no errors, the symbol FALSE otherwise. This will be the CLIPS return value of the `init` message in the absence of any other user-defined message-handlers.

**OTHER NOTES:** Implementation of the CLIPS function `init-slots`.

This function operates on the active instance. This function will normally be called as a result of `CoreInitializeInstance` sending an `init` message to an instance. This allows the user to define other message-handlers to perform actions before and after slot default initialization as described in the Basic Programming Guide.
The "initialization evaluation" flag for the instance (see the general notes in the Instance Commands Module) will be set if the appropriate prologue has been performed. The prologue is outlined in the description of \texttt{CoreInitializeInstance}. This function clears that flag to inform \texttt{CoreInitializeInstance} that the initialization is complete.

\textbf{FindInstanceBySymbol}

\textbf{PURPOSE:} Uses a hash table lookup to determine the address of the specified instance.

\textbf{ARGUMENTS:} The symbolic name of the instance.

\textbf{RETURNS:} The address of the instance, NULL if not found.

\textbf{FindInstanceSlot}

\textbf{PURPOSE:} Determines the address of the specified slot.

\textbf{ARGUMENTS:} 1) The address of the instance. 2) The symbolic name of the slot.

\textbf{RETURNS:} The address of the instance slot, NULL if not found.

\textbf{FindInstanceTemplateSlot}

\textbf{PURPOSE:} Uses a binary search on the symbolic hash value of the slot name to find the index of the specified slot in a class's instance slot template array.

\textbf{ARGUMENTS:} 1) The address of the class. 2) The symbolic name of the slot.

\textbf{RETURNS:} An integer index into the class's instance slot template array, -1 if not found.

\textbf{InitializeInstance}

\textbf{PURPOSE:} Initializes an instance. The descriptions of \texttt{ParseInitializeInstance} in the Instance Commands Module and \texttt{CoreInitializeInstance} give more details.

\textbf{ARGUMENTS:} A data object buffer for holding the result: the instance name on success or the symbol FALSE otherwise.

\textbf{OTHER NOTES:} Implementation of the CLIPS function \texttt{initialize-instance}. 
InitializeInstanceTable

PURPOSE: Support routine for SetupInstances in the Instance Commands Module which allocates and initializes the instance hash table.

MakeInstance

PURPOSE: Creates and initializes a new instance. The descriptions of ParseInitializeInstance in the Instance Commands Module and CoreInitializeInstance give more details.

ARGUMENTS: A data object buffer for holding the result: the instance name on success or the symbol FALSE otherwise.

OTHER NOTES: Implementation of the CLIPS function make-instance.

NoInstanceError

PURPOSE: Displays an error message when an instance cannot be found for a function call.

ARGUMENTS: 1) The name of the instance.
2) The name of the calling function.

PutSlotValue

PURPOSE: Stores a new value in a slot of an instance. All slot writing is passed through this central routine.

ARGUMENTS: 1) Address of the instance.
2) Address of the instance slot.
3) Data object holding the new slot value.
4) An integer code indicating whether to print watch messages for slot changes (1) or not (0).

RETURNS: An integer code indicating the result: SLOT_ERROR, SLOT_EMPTY or SLOT_FILLED (see Slot Value Set Codes).

OTHER NOTES: Old slot values are deinstalled and deallocated, and new slot values are installed. (De)installing a slot value means (de)incrementing the "in use" counts of all atoms in the data object. If the new value is a multifield, the a duplicate of the segment is assigned to the slot.
**QuashInstance**

**PURPOSE:** Removes an instance from its class's instance list and the global instance list and hash table. Also, all slot values are deinstalled (see **InstallInstance**) and erased. If the instance is not busy, it is deallocated, otherwise it is added to the instance garbage collection list (see **InstanceGarbageList**).

**ARGUMENTS:** A pointer to the instance.

**RETURNS:** A non-zero integer if the instance was successfully deleted, zero otherwise.

**OTHER NOTES:** The links going out from the instance to its class's list and the global class list and instance table are left unchanged; this allows outstanding pointers to this instance to still use it to follow links.

**SlotExistError**

**PURPOSE:** Prints out an appropriate error message when a slot cannot be found for a function.

**ARGUMENTS:** 1) The slot name.
2) The name of the calling function.

**SlotValueExpression**

**PURPOSE:** Support routine for **EvaluateDefaultSlots** in the Class Commands Module which generates an expression (or series of expressions in the case of a multifield) equivalent to a data object value for storage as a class slot default value.

**ARGUMENTS:** A data object address.

**RETURNS:** The equivalent expression(s).

**StaleInstanceAddress**

**PURPOSE:** Prints out an appropriate error message when an attempt is made to access an instance which is on the garbage collection list via a previously bound address.

**ARGUMENTS:** The name of the calling function.
ValidSlotValue

**PURPOSE:** Determines if a value is comprised of legal atoms for an instance slot. If it is not, the function generates an evaluation error and prints out an error message.

**ARGUMENTS:**
1) A data object pointer.
2) The name of the calling function.

**RETURNS:** A non-zero integer if the value is acceptable, zero otherwise.

**INTERNAL FUNCTIONS**

BuildDefaultSlots

**PURPOSE:** Support routine for BuildInstance which allocates an array of instance slots for a new instance.

**OTHER NOTES:** The new slots are attached to the instance indicated by the global `CurrentInstance`.

The address to hold the actual value of each slot is initialized according to the `shared` facet (see the general notes in the Instance Commands Module).

HashInstance

**PURPOSE:** Generates an index into the instance hash table for a given instance.

**ARGUMENTS:** The symbolic name of a instance.

**RETURNS:** The hash table index for the instance.

**OTHER NOTES:** The hash table index is derived from the symbol hash table index (see the Symbol Manager Module).

InsertSlotOverrides

**PURPOSE:** Support routine for CoreInitializeInstance which sends put messages for each slot-override.

**ARGUMENTS:**
1) A pointer to the instance.
2) A series of slot-override expressions.

**RETURNS:** A non-zero integer if successful, zero otherwise.

**OTHER NOTES:** The "slot override" flag is set for each slot with an override. These flags are later used by EvaluateInstanceSlots.
determine which slots need class default values and should be cleared by that function.

**InstallInstance**

**PURPOSE:** Support routine for BuildInstance which increments or decrements the "in use" counts of all atoms (e.g. symbols) associated with an instance, i.e the instance name and the slot values. This insures that all of these items persist at least as long as the instance does.

**ARGUMENTS:**
1) A pointer to an instance.
2) An integer code indicating whether to increment (1) or decrement (0) "in use" counts.

**InstanceLocationInfo**

**PURPOSE:** Support routine for BuildInstance which where a new instance belongs in the instance hash table.

**ARGUMENTS**
1) The symbolic name of the new instance.
2) A buffer for holding the address of the instance previous to the new one in the hash table.
3) A buffer for the hash value of the new instance.

**RETURNS:** A pointer to an old instance of the same name, NULL if none.

**InstanceSizeHeuristic**

**PURPOSE:** Support routine for CleanupInstances and QuashInstance which determines the amount of memory required by an instance. This amount is added or subtracted from a global count when the instance is added or removed from the instance garbage collection list respectively.

**ARGUMENTS:** The address of an instance.

**RETURNS:** The amount of memory required by the instance.

**OTHER NOTES:** Implemented as a preprocessor macro.

CLIPS normally allows garbage memory to accumulate to a certain level before bothering to release it back to the system. This drastically improves performance.

**NewInstance**

**PURPOSE:** Support routine for BuildInstance which allocates a new instances data structure and initializes all the fields.
RETURNS: The address of a new instance.

StoreValuesInMultifield

PURPOSE: Support routine for EvaluateAndStoreInDataObject which creates a multifield and stores a series of values in it.

ARGUMENTS: 1) A series of atomic data objects chained together via their "next" fields.
2) A data object buffer to hold the resulting multifield.
3) The number of data objects in the source list.
The Message-Handler Commands Module (msgcom.c) contains the parsing and general interface routines for the procedural attachments (message-handlers) to classes. For a description of the defmessage-handler construct, see the Basic Programming Guide. The defmessage-handler construct capability, along with the other features of the CLIPS Object-Oriented Language (COOL), can be removed by using the appropriate compile flag in the setup header file. The message-handler data structure is summarized in the following diagram:

<table>
<thead>
<tr>
<th>Name (Symbol Pointer)</th>
<th>Class (Class Pointer)</th>
<th>Busy Count (int)</th>
<th>Bitfields (stored as 1 integer)</th>
<th>Minimum Parameters (int)</th>
<th>Maximum Parameters (int)</th>
<th>Actions (Expression Pointer)</th>
<th>Pretty-Print Form (array of char)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram of Message-Handler Data Structure]

The internal data structure of a defmessage-handler construct primarily consists of: a symbolic name; a pointer to the parent class; two integers, which indicate the minimum and maximum number of arguments the handler will accept respectively; and a sequence of expressions which comprise the body of the handler. If a handler has a wildcard parameter (i.e. the handler will accept any number of arguments greater than or equal to the minimum number of arguments), the maximum number of arguments field will have the value -1. A busy count for each handler reflects how many times a handler is executing on behalf of a message. This busy must be zero before any of the handlers of the class to which this handler belongs can be modified. Other fields in the handler data structure include: the pretty-print form and bitfields. The bitfields are stored in a single integer and indicate the following information about a handler: whether the handler is one of the predefined ones attached to the USER class, the type code (see Message-Handler Type Codes in the Message-Handler Functions Module) and whether the handler has been marked for deletion.

When a handler is called during a message dispatch (see the Message-Handler Functions Module), if the number of arguments is outside the acceptable range, the entire message is immediately terminated and an error is generated. Otherwise, all the actions of the handler are evaluated in order as if they were grouped in a progn. The evaluation of the last expression in the handler body is returned as the value of the handler, unless an error occurs or the return function is used (see the Basic Programming Guide).
The arguments of a handler are evaluated and stored in order in an array of data objects called the message parameter array (CurrentMessageFrame). Variable references within the body of a handler are replaced when the construct is loaded with function calls which either access the bind list (see the Primary Functions Module), get the value of a global variable (see the Defglobal Manager Module) or positionally access the message parameter array (HandlerRtnUnknown). For example, references to the second parameter of a handler are replaced with function calls which access the second data object in the parameter array at run-time.

A wildcard parameter allows the handler to accept any number of arguments. All references to the wildcard parameter are replaced with a call to a special function, HandlerWildargs, which groups all of the data objects in the parameter array starting at the position of the wildcard parameter to the end of the array into a multifield data object.

If a parameter (including a wildcard parameter) is rebound anywhere within the body of the deffunction, all references to that parameter are replaced with calls to a special function, HandlerGetBind, which first checks the bind list before accessing the parameter array.

GLOBAL VARIABLES

<table>
<thead>
<tr>
<th>DELETE_STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
<tr>
<td>OTHER NOTES:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DELETE_SYMBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INIT_STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
<tr>
<td>OTHER NOTES:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INIT_SYMBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE:</td>
</tr>
</tbody>
</table>
INTERNAL VARIABLES

**PRINT_STRING**

PURPOSE: Name of the predefined system message-handler attached to the USER class which pretty-prints an instance.

OTHER NOTES: Implemented as a preprocessor constant.

**SELF_LEN**

PURPOSE: Length of SELF_STRING.

OTHER NOTES: Implemented as a preprocessor constant.

**SELF_SLOT_REF**

PURPOSE: The string used to attach a direct slot reference to active instance parameter in a message-handler.

OTHER NOTES: Implemented as a preprocessor constant.

**SELF_STRING**

PURPOSE: Lexeme for SELF_SYMBOL.

OTHER NOTES: Implemented as a preprocessor constant.

**SELF_SYMBOL**

PURPOSE: The symbol used to represent the active instance parameter.

GLOBAL FUNCTIONS

**AddSystemHandlers**

PURPOSE: Support routine for SetupMessageHandlers which defines the three system message-handlers for initialization, deletion and printing and attaches them to the USER class.

**CheckHandlerAgainstSlots**

PURPOSE: Support routine for AddClass in the Class Functions Module and ParseDefmessageHandler which insures that a message-handler does not conflict with the implicit handlers (slot-accessors) for a class. For example, if a class has a slot called bar, it is illegal to define an explicit primary handler called get-bar or put-bar.
ARGUMENTS: 1) A pointer to a class.
2) The symbolic name of the message-handler.
3) An integer code representing the type of the handler (see Message-Handler Type Codes in the Message-Handler Functions Module).

RETURNS: A non-zero integer if the handler does not conflict with any of the slots, zero otherwise.

OTHER NOTES: The new handler will be illegal only if it is primary and it conflicts with one of the direct slots of a class. Primary handlers with same name as an inherited slot will shadow the slot-accessor. The Basic Programming Guide gives a complete explanation of handler shadowing.

**CmdListDefmessageHandlers**

PURPOSE: Lists message-handlers for a class.

OTHER NOTES: Implementation of the CLIPS function list-defmessage-handlers.

**CmdUndefmessageHandler**

PURPOSE: Removes a message-handler from a class.

OTHER NOTES: Implementation of the CLIPS function undefmessage-handler.

**Embedded Access for Defmessage-Handlers**

PURPOSE: The following functions are provided for embedded access and are documented in the Advanced Programming Guide: DeleteDefmessageHandler, FindDefmessageHandler, GetDefmessageHandlerName, GetDefmessageHandlerPPForm, GetDefmessageHandlerType, GetNextDefmessageHandler, IsDefmessageHandlerDeletable, ListDefmessageHandlers, PreviewMessage and WildDeleteHandler.

OTHER NOTES: There are additional embedded access functions for message-handlers in the Message-Handler Functions Module.
**GroupHandlerWildargs**

**PURPOSE:** Stores the message parameter array elements from the specified beginning index minus one to the end of the array in the caller's multifield data object.

**ARGUMENTS:**
1) A pointer to a data object to hold the resulting multifield value.
2) The index (one is the beginning) from which to start copying the parameter array.

**HandlerDeleteSlot**

**PURPOSE:** Deletes fields from a multifield slot value of the active instance.

**RETURNS:** A non-zero integer if the slot was modified successfully, zero otherwise.

**OTHER NOTES:** Implementation of the CLIPS function `direct-mv-delete`.

**HandlerGetBind**

**PURPOSE:** Determines the value of a specified variable reference within the body of a handler. The symbolic name of the variable and an index indicating if the variable is a handler parameter are CLIPS supplied arguments. If the variable is on the bind list, that value is returned. Otherwise, the value of the parameter specified by the index is returned. In the event that the variable is neither on the bind list nor is it a parameter, an error will be generated.

**ARGUMENTS:** A pointer to a data object which will hold the value of the bound variable.

**OTHER NOTES:** Implementation of the internal CLIPS function `hndgetbind`.

Used for general variable references, including bind list variables and handler parameters which are rebound within the actions of the handler. If the index is zero, the variable is not a handler parameter. The absolute value of the index minus one is the position of the parameter in the message parameter array. If the index is less than zero, the variable corresponds to the wildcard parameter.
**HandlerGetSlot**

Purpose: Directly reads the slot specified by the CLIPS supplied argument of the active instance.

Arguments: Data object buffer for holding the slot value.

Other Notes: Implementation of the CLIPS function `get`. Direct slot references in a handler are replaced with calls to this function when the handler is parsed (see `ReplaceHandlerParameters`).

**HandlerInsertSlot**

Purpose: Inserts fields into a multifield slot value of the active instance.

Returns: A non-zero integer if the slot was modified successfully, zero otherwise.

Other Notes: Implementation of the CLIPS function `direct-mv-insert`.

**HandlerPutSlot**

Purpose: Directly writes a value to the slot (both specified by CLIPS supplied arguments) of the active instance.

Returns: A non-zero integer if the write was successful, zero otherwise.

Other Notes: Implementation of the CLIPS function `put`.

**HandlerReplaceSlot**

Purpose: Replaces fields in a multifield slot value of the active instance.

Returns: A non-zero integer if the slot was modified successfully, zero otherwise.

Other Notes: Implementation of the CLIPS function `direct-mv-replace`.

**HandlerRtnUnknown**

Purpose: Gets the value of the specified element of the message parameter array, where the element index plus one is given as a CLIPS supplied argument.
ARGUMENTS: A pointer to a data object which will hold the value of the bound variable.

OTHER NOTES: Implementation of the internal CLIPS function (hndunknown).

Used for references to regular handler parameters which are never rebound within the actions of the handler.

**HandlerWildargs**

PURPOSE: Gets the values of the specified elements of the message parameter array and groups them into a multifield data object, where the range of elements is given by a CLIPS supplied argument minus one to the end of the message parameter array.

ARGUMENTS: A pointer to a data object which will hold the value of the bound variable.

OTHER NOTES: Implementation of the internal CLIPS function (hndwildargs).

Used for references to a wildcard handler parameter which is never rebound within the actions of the handler.

**PPDefmessageHandler**

PURPOSE: Displays the pretty-print form of a message-handler.

OTHER NOTES: Implementation of the CLIPS function ppdefmessage-handler.

**PreviewMessageCmd**

PURPOSE: Displays all the applicable message-handlers for a particular send call. The message and arguments are supplied by CLIPS.

OTHER NOTES: Implementation of the CLIPS function preview-send.

**SetupMessageHandlers**

PURPOSE: Defines all functions and commands for the defmessage-handler construct. Sets up all necessary load and watch interfaces.

OTHER NOTES: Initialization differs between standard and run-time configurations.
INTERNAL FUNCTIONS

**CheckHandlerBindList**

**PURPOSE:** Support routine for ParseDefmessageHandler which insures that a message-handler makes no attempt to rebind the active instance parameter or direct slot references.

**RETURNS:** A no-zero integers if all the binds were legal, zero otherwise.

**FindHandlerParameter**

**PURPOSE:** Support routine for ReplaceHandlerParameters which determines the position of a particular parameter in the list of all message-handler parameters.

**ARGUMENTS:**
1) The symbolic name of a parameter.
2) The list of parameters parsed so far.

**RETURNS:** The integer zero if the named parameter is not already in the list, otherwise the position of the parameter in the list.

**InsertHandlerHeader**

**PURPOSE:** Support routine for ParseDefmessageHandler and NewSystemHandler which appends a new handler into the array of handlers for a class.

**ARGUMENTS:**
1) A pointer to the class.
2) The symbolic name of the handler.
3) An integer code representing the type of the handler (see Message-Handler Type Codes in the Message-Handler Functions Module).

**RETURNS:** A pointer to the new handler.

**OTHER NOTES:** This routine also creates a map of integer indices into the handler array which gives the order according to the hash values of the symbolic names of the handlers. Thus, handlers for a class can be easily found by performing a binary search on the symbolic hash value of the handler name.

**NewSystemHandler**

**PURPOSE:** Support routine for AddSystemHandlers which adds a new message-handler to the handler array of the USER class. A system handler is assumed to be of type primary. The handler has zero or one explicit parameters, and the
body contains one function call which either takes no arguments or takes the one explicit handler parameter as an argument.

ARGUMENTS:
1) Name of the system class.
2) Name of the system message-handler.
3) Name of the CLIPS function called in the body of this handler.
4) An integer code indicating if argument #3 requires a parameter (1) or not (0).
5) The address of a CLIPS function for accessing handler parameters (see HandlerRtnUnknown).

OTHER NOTES: In CLIPS 5.1, there are no system handlers which require the use of a handler parameter. Thus, the third and fourth arguments to this function are unnecessary and will be eliminated in the next release.

The CLIPS syntax for the three system message-handlers are:

(defmessage-handler USER init primary ()
  (init-slots))

(defmessage-handler USER delete primary ()
  (delete-instance))

(defmessage-handler USER print primary ()
  (ppinstance))

---

**ParseDefmessageHandler**

**PURPOSE:**
Used by the load command to parse a defmessage-handler.

**ARGUMENTS:**
The logical name of the input source.

**RETURNS:**
The integer zero if there are no parsing errors, non-zero otherwise.

**OTHER NOTES:**
Installation of the message-handler symbolic name and action expressions is handled by this routine; deinstallation is handled by DeallocateMarkedHandlers in the Message-Handler Functions Module or InstallClass in the Class Functions Module.

---

**ParseHandlerParameters**

**PURPOSE:**
Support routine for ParseDefmessageHandler which parses a message-handler parameter list.
ARGUMENTS: 1) The logical name of the input source.
2) Buffer for wildcard symbol (if any).
3) Buffer to hold scanned tokens.

RETURNS: A series of expressions containing the parameter names, NULL on errors.

OTHER NOTES: The active instance parameter is prepended to the list of parameters.

This routine insures that there are no duplicate parameters or parameters which look like direct slot references.

---

**ReplaceHandlerParameters**

PURPOSE: Support routine for ParseDefmessageHandler which replaces all variable references in the message-handler actions with appropriate function calls that access the bind list, the message parameter array or global variables at runtime. Direct slot references are replaced with calls to the function get.

ARGUMENTS: 1) The list of action expressions.
2) The list of parameter name expressions.
3) The symbolic name of a wildcard parameter (if any).

RETURNS: The integer zero if there are no parsing errors, non-zero otherwise.
The Message-Handler Functions Module (msgfun.c) implements the message dispatch when a message is actually sent to an object and maintains the internal details of the defmessage-handler construct. For a description of the defmessage-handler construct, see the Basic Programming Guide. The defmessage-handler construct capability, along with the other features of the CLIPS Object-Oriented Language (COOL), can be removed by using the appropriate compile flag in the setup header file.

When a message is dispatched with the `send` function, CLIPS creates the message parameter array, which is comprised of the message object and arguments, and forms a list of all the applicable methods to the message. The handlers in this list are linked using a temporary data structure called a handler link:

![Handler Link Diagram]

A handler contains: a generic pointer, which is a pointer to an explicit defmessage-handler or an instance slot if the handler is a slot-accessor, bitfields and a pointer to the next handler link. The bitfields indicate the following: whether the handler is a slot-accessor (i.e. implicit); the type of the handler (see Message-Handler Type Codes); and, if the handler is a slot-accessor, whether it is for reading or for writing. The details of forming the list of applicable handlers are given in the description of the function `FindApplicableHandlers`. The mechanics of a message dispatch are outlined in the description of the function `DispatchMessage`.

GLOBAL VARIABLES

**CurrentMessageFrame**

**Purpose:** A pointer to an array of data objects which are the evaluated arguments for the currently executing message.

**CurrentMessageName**

**Purpose:** A symbol indicating the name of the currently executing message used for error and trace messages.

**CurrentMessageSize**

**Purpose:** An integer indicating the number of data objects in the currently executing message's parameter array.
hndquals

PURPOSE: An array of strings giving the textual descriptions of the message-handler types (see Message-Handler Type Codes). A handler type is an integer code, and the string corresponding to a type n is in the nth position of the array, e.g. hndquals[MAROUND] is "around". This array is used in printing out trace and error messages as well as parsing handler types.

Message-Handler Lookup Codes

PURPOSE: The integer codes LOOKUP_HANDLER_ADDRESS and LOOKUP_HANDLER_INDEX are used by the function FindHandler to determine whether to return message-handler addresses or indices into the class's handler array.

OTHER NOTES: Implemented as preprocessor constants in msgfun.h.

Message-Handler Type Codes

PURPOSE: MAFTER, MAROUND, MBefore, MPRIMARY and MERROR (see the description of around, before, after and primary handlers in the Basic Programming Guide).

OTHER NOTES: Implemented as preprocessor constants in msgfun.h.

Slot-Accessor Prefix Strings

PURPOSE: GSM_PREFIX is the string prepended to all slot names to yield the name of the read slot-accessor, and GSMP_LEN is the length of that string. PSM_PREFIX is the string prepended to all slot names to yield the name of the write slot-accessor, and PSMP_LEN is the length of that string.

OTHER NOTES: Implemented as preprocessor constants in msgfun.h.

WatchHandlers

PURPOSE: An integer flag indicating whether or not to print out trace information whenever a message-handler begins and ends execution. This flag is used by the watch command.

WatchMessages

PURPOSE: An integer flag indicating whether or not to print out trace information whenever a message begins and ends execution. This flag is used by the watch command.
INTERNAL VARIABLES

CurrentCore

PURPOSE: A handler link to the currently executing message-handler.

Message Trace Strings

PURPOSE: BEGIN_TRACE and END_TRACE are the strings used in trace printouts to indicate the beginning and end of execution of a message or a message-handler.

OTHER NOTES: Implemented as preprocessor constants.

NextInCore

PURPOSE: A handler link to the next applicable message-handler after the currently executing handler.

PERFORM

PURPOSE: Unused preprocessor constant which will be removed in the next release.

PREVIEW

PURPOSE: Unused preprocessor constant which will be removed in the next release.

TopOfCore

PURPOSE: A handler link to the applicable message-handler with the highest precedence. The list is in order according to message-handler precedence.

GLOBAL FUNCTIONS

CallNextHandler

PURPOSE: Executes message-handlers shadowed by the currently executing handler. This function can only be called from the actions of the currently executing message-handler.

ARGUMENTS: A pointer to a data object to store the return value of the shadowed handlers.

OTHER NOTES: Implementation of the CLIPS functions call-next-handler and override-next-handler.
Following is a summary of CallNextHandler:

1. Immediately return with an error in all but the following scenarios:
   1a) The currently executing handler (CurrentCore) is of type around, and there is at least one shadowed handler available, or
   1b) The currently executing handler is of type primary, and the next available handler (NextInCore) is also of type primary.

2. If override-next-handler has been called, save the old message parameter array, and create a new one based on the function arguments.
3. Save the state of the bind list and then destroy it.
4. Save the values of CurrentCore and NextInCore and then advance them ahead one in the list of applicable handlers.
5. If the currently executing handler is of type primary, go to step 7.
6. If the next available handler is of type around, call EvaluateExpression for the actions of that handler and capture the result. Otherwise, call CallHandlers for the remaining core of handlers and capture the result. Go to step 8.
7. If the next available handler is not a slot-accessor, call EvaluateExpression for the actions of that handler and capture the result. Otherwise, call PerformImplicitHandler to execute the slot-accessor and capture the result.
8. Restore the previous bind list, the values of CurrentCore and NextInCore and the message parameter array (if necessary).

### CheckCurrentMessage

**PURPOSE:** Insures that a message is currently executing for functions which operate on the active instance.

**ARGUMENTS:**
1) Name of the calling function.
2) An integer flag indicating if the function can operate on primitive type objects (0) or only instances of user-defined classes (1).

**RETURNS:** A non-zero integer if the active instance is valid, zero otherwise.
**DeallocateMarkedHandlers**

**PURPOSE:** Support routine for **ClearDefclasses** and **AddClass** in the Class Functions Module as well as **DeleteDefmessageHandler** in the Message Commands Module and **DeleteHandler** which removes marked handlers from a class's handler array.

**ARGUMENTS:** A pointer to the class.

**OTHER NOTES:** The "mark" fields of the handlers are assumed to have been set by the calling function.

A symbolic hash value sorted map of handlers is also kept with the class (see the general notes in the Class Commands Module). In order to update this map after the deletion of handlers, the "busy" field of a handler is temporarily used to count how many handlers before it in the array will be deleted. That handlers position in the map is then adjusted accordingly.

**DeleteHandler**

**PURPOSE:** Support routine for **WildDeleteHandler** which removes one or more handlers from a class.

**ARGUMENTS:**
1) A pointer to the class.
2) The symbolic name of the handler. If the name is "***", and there is no handler named "***" in the class, then all handlers matching the type will be deleted.
3) An integer code representing the type of the handler (see **Message-Handler Type Codes**). If the type is -1, then all message-handlers matching the name will be deleted.
4) An integer flag indicating whether to print error messages when matching handlers cannot be found (1) or not (0).

**RETURNS:** A non-zero integer if the handlers are successfully deleted, zero otherwise.

**DestroyHandlerLinks**

**PURPOSE:** Support routine for **PreviewMessage** in the Message Commands Module and **PerformMessage** which deallocates the temporary links between a core of handlers applicable to a message.

**ARGUMENTS:** A pointer to the top of the temporary handler links.
OTHER NOTES: The "busy" counts of the handlers and their classes, which were incremented by FindApplicableHandlers, are decremented.

**DirectMessage**

**PURPOSE:** Support routine for functions in the Instance Commands and Functions Modules which sends a message to an object, e.g. the init message in a make-instance call.

**ARGUMENTS:**
1) The first part of the message name in the form of a string (can be NULL).
2) The second part of the message name in the form of a symbol.
3) A pointer to an instance.
4) A data object buffer for storing the result of the message (can be NULL if irrelevant).
5) A series of expressions representing the message arguments.

**OTHER NOTES:** The message name is broken into two arguments so that slot-accessor messages can easily be formed from slot names. However, the slot-accessor symbol names are stored with the slot, so the symbol could be passed directly rather than requiring that this routine construct them. This routine will be enhanced in the next release.

**DispatchMessage**

**PURPOSE:** This routine is called by EvaluateExpression in the Evaluation Module to process a message. The message name, object and arguments are supplied by CLIPS. The message dispatch is described in detail in the Basic Programming Guide.

**ARGUMENTS:** A data object buffer to hold the result of the message.

**OTHER NOTES:** Implementation of the CLIPS function send.

This function determines the symbolic name of the message from the CLIPS arguments and prepends the message object expression to the other message arguments. The bulk of the message dispatch is done by the routines PerformMessage, CallHandlers and CallNextHandler.
**DisplayCore**

**PURPOSE:** Support routine for **PreviewMessage** in the Message Commands Module which recursively displays the set of applicable handlers for a particular message.

**ARGUMENTS:**
1) A list handler links to applicable handlers.
2) The level of indentation indicating the depth of handler shadowing.

**Embedded Access for Defmessage-Handlers**

**PURPOSE:** The function **CLIPSSendMessage** is provided for embedded access and is documented in the *Advanced Programming Guide*.

**OTHER NOTES:** There are additional embedded access functions for message-handlers in the Message-Handler Commands Module.

**FindHandler**

**PURPOSE:** Support routine for **ParseDefmessageHandler** and others in the Message Commands Module and **DeleteHandler** which looks up a message-handler.

**ARGUMENTS:**
1) A pointer to a class.
2) The symbolic name of the handler.
3) An integer code representing the type of the handler (see **Message-Handler Type Codes**).
4) An integer code indicating for the return value to be a handler address or an index into the class's handler array (see **Message-Handler Lookup Codes**).

**RETURNS:** If a handler address was requested, a generic pointer is returned which is the handler address, or NULL if not found. Otherwise, an integer typecast into a generic pointer is returned which is the index of the handler in the class's handler array, or -1 if not found.

**FindHandlerNameGroup**

**PURPOSE:** Support routine for **ListDefmessageHandlers** in the Message Commands Module, **FindHandler** and **FindApplicableOfName** which performs a binary search on the symbolic hash value of the handler name to find a group of handlers which names all have the same hash value as the given name.
ARGUMENTS: 1) A pointer to a class.
2) The symbolic name of a handler.

RETURNS: An index into the sorted hash value map of the class's handler array where handlers which names have the same hash value as the given name begin, -1 if there are none. The actual given name must be present in the group (e.g. they could all be handlers which names just happened to have the same hash value as the given name), or the return value will still be -1.

**FindPreviewApplicableHandlers**

PURPOSE: Support routine for **PreviewMessage** in the Message Commands Module which generates a ranked list of applicable message-handlers for a message.

ARGUMENTS: 1) A pointer to a class.
2) The symbolic name of the message.

RETURNS: The top of a list of temporary handler links forming the core of applicable handlers.

OTHER NOTES: This function differs from **FindApplicableHandlers** in that it uses **FindClassSlot** to find slot-accessor handlers rather than **FindInstanceSlot**.

**HandlerDeleteError**

PURPOSE: Support routine for **ClearDefclasses** in the Class Functions Module, **DeleteDefmessageHandler** in the Message Commands Module and **DeleteHandler** which prints out an error message when a message-handler cannot be deleted from a class.

ARGUMENTS: The name of the class.

**HandlersExecuting**

PURPOSE: Support routine for message-handler parsing and deletion routines which determines if any handlers attached to a class are currently executing.

ARGUMENTS: A pointer to the class.

RETURNS: A non-zero integer if any of the handlers of the class are executing, zero otherwise.
**HandlerType**

**PURPOSE:** Support routine for message-handler parsing and access routines which determines the integer code for a handler type given the string representation (see Message-Handler Type Codes).

**ARGUMENTS:**
1) The name of the calling function.
2) The string representation of the handler type, e.g. "primary"

**RETURNS:** The handler type code.

**NewHandler**

**PURPOSE:** This function is not used in CLIPS 5.1 and will be deleted in the next release.

**NextHandlerAvailable**

**PURPOSE:** Determines if a shadowed message-handler is available for execution by call-next-handler or override-next-handler. See the description of CallNextHandler for details.

**RETURNS:** A non-zero integer if a shadowed handler is available, zero otherwise.

**OTHER NOTES:** Implementation of the CLIPS function next-handlerp.

**PrintAbbreviatedHandlerRemoval**

**PURPOSE:** Support routine for DeleteClass in the Class Functions Module and DeallocateMarkedHandlers which a brief description of a message-handler being removed.

**ARGUMENTS:** A pointer to the handler.

**PrintCurrentMessage**

**PURPOSE:** Support routine for HandlerGetBind in the Message Commands Module which prints a synopsis of the currently executing message for unbound variable errors.

**ARGUMENTS:** Logical name of the output destination.
PrintHandler

PURPOSE: Support routine for DisplayHandlersInLinks in the Class Commands Module, ListDefmessageHandlers in the Message Commands Module, PrintCurrentMessage, PrintPreviewHandler and TraceHandler which displays a brief description of a message-handler.

ARGUMENTS: 1) Logical name of the output destination. 2) Name of the class. 3) Name of the handler. 4) Handler type string.

INTERNAL FUNCTIONS

CallHandlers

PURPOSE: Support routine for PerformMessage and CallNextHandler which executes all the before, primary and after message-handlers applicable to a message.

ARGUMENTS: A data object buffer to hold the result of executing the most specific primary handler.

OTHER NOTES: Following is a summary of CallHandlers:

1) Save the state of the bind list and then destroy it.
2) Save the values of CurrentCore and NextInCore.
3) Call EvaluateExpression for the actions of each before handler in order and advance CurrentCore and NextInCore appropriately.
4) Call EvaluateExpression for the actions of the first primary handler, capture the result and advance CurrentCore and NextInCore to skip over any other primary handlers. Other primary handlers are shadowed by the first, and call-next-method must be used within the body of the first primary handler to execute them.
5) Call EvaluateExpression for the actions of each after handler in order and advance CurrentCore and NextInCore appropriately.
6) Restore the bind list and the old values of CurrentCore and NextInCore.

The bind list is reset and ReturnFlag is cleared after the execution of each handler.
CheckHandlerArgCount

PURPOSE: Support routine for PerformMessage, CallHandlers and CallNextHandler which verifies that the current message parameter array satisfies the current handler's parameter count restriction.

RETURNS: A no-zero integer if the number of arguments is satisfactory, zero otherwise.

DisplayPrimaryCore

PURPOSE: Support routine for DisplayCore which recursively displays the set of applicable primary handlers for a particular message.

ARGUMENTS: 1) A list of handler links to applicable handlers. 2) The level of indentation indicating the depth of handler shadowing.

RETURNS: The handler link to the handler immediately following the primary handlers (if any).

EvaluateMessageParameters

PURPOSE: Support routine for CallNextHandler and PerformMessage which evaluates all the CLIPS supplied argument expressions for a message and stores the resulting values in the message parameter array (CurrentMessageFrame).

ARGUMENTS: 1) The list of parameter name expressions. 2) The number of parameters.

RETURNS: A pointer to an array of data objects containing the evaluations of the message argument expressions.

FindApplicableHandlers

PURPOSE: Support routine for PerformMessage which generates a ranked list of applicable handlers for a message.

ARGUMENTS: 1) A pointer to a class. 2) The symbolic name of the message.

RETURNS: The top of a list of temporary handler links forming the core of applicable handlers.
OTHER NOTES: The "related symbol" field of the message name is used to access the slot name symbol when checking if a slot-accessor handler is applicable to a message. A link in the core formed by this routine can point to an explicit defmessage-handler or an instance slot in the case of a slot-accessor.

A handler is applicable to a message if its name matches that of the message and it is attached to one of the classes of which the message object is an instance. All the applicable handlers are inserted into a "core" of applicable messages ordered in the following way:

1. All around handlers from the most specific class of the message object to the most general.
2. All before handlers from the most specific class of the message object to the most general.
3. All primary handlers from the most specific class of the message object to the most general.
4. All after handlers from the most general class of the message object to the most specific.

This ordering is accomplished by forming three queues for the around, before and primary handlers respectively and one stack for the after handlers. The class precedence list of the class of the message object is then examined in order from most specific to most general. The support routine FindApplicableOfName takes care of appending all applicable around, before and primary handlers from a class to the appropriate queues and pushing an applicable after handler onto the stack. When all classes have been processed, the support routine JoinHandlerLinks forms the core of applicable handlers by simply linking the three queues and one stack together.

The "busy" counts are incremented for each applicable handler and the class to which it belongs.

---

**FindApplicableOfName**

**PURPOSE:** Support routine for FindPreviewApplicableHandlers and FindApplicableHandlers which adds applicable handlers for a class to the handler type queues and stack.

**ARGUMENTS:**

1) A pointer to a class.
2) An array of pointers to the tops of the handler link type queues and stack.
3) An array of pointers to the bottoms of the handler link type
queues and stack.
4) The symbolic name of the message.

RETURNS: A no-zero integer if any applicable primary handlers were found for the class, zero otherwise.

JoinHandlerLinks

PURPOSE: Support routine for FindPreviewApplicableHandlers and FindApplicableHandlers which handler type queues and stack together to form the final ordered core of applicable message-handlers.

ARGUMENTS: 1) An array of pointers to the tops of the handler link type queues and stack.
2) An array of pointers to the bottoms of the handler link type queues and stack.
3) The symbolic name of the message.

RETURNS: The core list of applicable handlers, NULL on errors.

OTHER NOTES: If there are no applicable primary handlers, this routine deletes the queues and stack and issues an error message.

PerformImplicitHandler

PURPOSE: Support routine for CallNextHandler and CallHandlers which handles the execution of slot-accessor handlers (i.e. get- and put- messages).

ARGUMENTS: A data object buffer to hold the result of the slot access.

PerformMessage

PURPOSE: Support routine for CLIPSSendMessage, DirectMessage and DispatchMessage which is the main driver for a message dispatch.

ARGUMENTS: 1) A data object buffer to hold the result of the message.
2) A series of message argument expressions/
3) The symbolic name of the message.

OTHER NOTES: Following is a summary of PerformMessage:

1. Save previous values of globals, such as CurrentMessageName and TopOfCore, and set them for the new message.
2. Save the state of the bind list and then destroy it.
3. Save the states of the return and break contexts and set
them to FALSE.
4. Increment the evaluation depth (see the Evaluation Module).
5. Count and evaluate the arguments and store them in the message parameter array. The message object will always be the first element in the parameter array.
6. Increment the "busy" count of the message object if it is an instance of a user-defined class.
7. Call **FindApplicableHandlers** to determine the set of applicable handlers for the message.
8. If the first available handler is of type **around**, call **EvaluateExpression** for the actions of that handler and capture the result. Otherwise, call **CallHandlers** for the core of handlers and capture the result.
9. Deallocate the core of applicable handlers.
10. Restore all global values to their previous states.
11. Decrement the evaluation depth.
12. Clear **ReturnFlag**.
13. Adjust the evaluation depth of the return value (see **PropogateReturnValue** in the Evaluation Module).

---

**PrintNoHandlerError**

**PURPOSE:** Support routine for **CLIPSSendMessage** and **JoinHandlerLinks** which prints out an error message when no applicable **primary** handlers can be found for a message dispatch.

**ARGUMENTS:** The name of the message.

**PrintPreviewHandler**

**PURPOSE:** Support routine for **DisplayCore** and **DisplayPrimaryCore** which prints a synopsis of a handler.

**ARGUMENTS:**
1) A handler link in the list of applicable handlers.
2) The level of indentation indicating the depth of handler shadowing.
3) A string indicating the beginning or end of execution of a handler.

**TraceHandler**

**PURPOSE:** Used by the **watch** command to print out trace messages when a message-handler begins and ends execution.

**ARGUMENTS:**
1) The logical name of the output destination.
2) A handler link in the list of applicable handlers.
3) A string indicating the beginning or end of execution of a handler.

### TraceMessage

**PURPOSE:** Used by the watch command to print out trace messages when a message begins and ends execution.

**ARGUMENTS:**
1) The logical name of the output destination.
2) A string indicating the beginning or end of execution of a message.
The Instance-Set Queries Module (insquery.c) provides the routines for a useful query system which can determine and perform actions on sets of instances of user-defined classes that satisfy user-defined criteria.

Above is an example excerpted from the Basic Programming Guide which shows a complete instance-set query function call. The instance-set template is internally represented by a series of data structures called Query Class Restrictions. The structure is summarized by following diagram:

The fields are: a pointer to a class, a pointer to the next class restriction list and a pointer to the next class in the current restriction list. Thus, for the example above, the instance-set template would look like:
While an instance-set query function is executing, it uses a **Query Core** data structure to hold information about the query and instance-sets which satisfy the query. The data structure is summarized in the following diagram:

```plaintext
Query Core
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Query (Expression Pointer)</td>
</tr>
<tr>
<td>Action (Expression Pointer)</td>
</tr>
<tr>
<td>Action Result (Data Object Pointer)</td>
</tr>
<tr>
<td>Instance-Set Solution (Instance Pointer Array)</td>
</tr>
<tr>
<td>Instance-Set Size (int)</td>
</tr>
<tr>
<td>Solution Set Top (Instance Pointer Array)</td>
</tr>
<tr>
<td>Solution Set Bottom (Instance Pointer Array)</td>
</tr>
<tr>
<td>Solution Count (int)</td>
</tr>
</tbody>
</table>
```

The fields are: the expression that must be satisfied for generated instance-sets, the action that will be performed for instance-sets which satisfy the query, a data object buffer to hold the results of evaluating the action, an intermediary array of instance pointers to hold the generated instance-sets, the number of instances in an instance-set, the top and bottom of a list of arrays of instance pointers to save instance-sets which satisfy the query and the number of instance-sets in the solution list. Actions are used only for: **do-for-instance**, **do-for-all-instances** and **delayed-do-for-all-instances**. The solution instance-sets need only be saved for **find-all-instances** and **delayed-do-for-all-instances**.

Instance-set query functions can be nested and can access variables outside their scope, including the member variables of other instance-set query functions in which they are nested. Thus, each executing instance-set query function must have its own unique query core. These cores are stored in a stack of **Query Stack Nodes**. The data structure is summarized in the following diagram:

```plaintext
Query Core Stack Node
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Core (Query Core Pointer)</td>
</tr>
<tr>
<td>Next Link (QCSN Pointer)</td>
</tr>
</tbody>
</table>
```

The fields are: a pointer to a query core and a pointer to the next query stack node.

A detailed description of how instance-set query functions are parsed is given in **ParseQueryAction**, **ParseQueryNoAction** and their associated functions. The mechanics of processing an instance-set query are given in **TestEntireChain**, **TestEntireClass**, **TestForFirstInChain**, **TestForFirstInChain** and **TestForFirstInstanceInClass** and their associated functions.
GLOBAL VARIABLES

<table>
<thead>
<tr>
<th>BITS_PER_BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: The number of bits in a byte. Used to determine the number of bytes necessary to store the traversal map for a class (see the general notes on the Class Commands Module).</td>
</tr>
<tr>
<td>OTHER NOTES: Implemented as a preprocessor constant in object.h.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>QUERY_DELIMITER_STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Lexeme for QUERY_DELIMITER_SYMBOL.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>QUERY_DELIMITER_SYMBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Symbol used to mark the ends of class restriction lists in the parsed form of an instance-set query function (see the functions ParseQueryAction and ParseQueryNoAction).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAX_TRAVERSALS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: The maximum number of times a single class can be examined by simultaneous hierarchy traversals (see the general notes on the Class Commands Module and also see the descriptions of CTID, GetTraversalID, SetTraversalID, TestTraversalID, and ReleaseTraversalID).</td>
</tr>
<tr>
<td>OTHER NOTES: Implemented as a preprocessor constant in object.h.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TRAVERSAL_BYTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: The number of bytes necessary to store the traversal map for a single class (see the general notes on the Class Commands Module).</td>
</tr>
<tr>
<td>OTHER NOTES: Implemented as a preprocessor constant in object.h.</td>
</tr>
</tbody>
</table>

INTERNAL VARIABLES

<table>
<thead>
<tr>
<th>AbortQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: An integer flag which is set when no instances are found for a particular instance-set template member which satisfy the query. When this flag is set, the instance-set query function being processed will be immediately terminated.</td>
</tr>
</tbody>
</table>
**CTID**

PURPOSE: The next available integer identifier for a class hierarchy traversal (see the general notes of the Class Commands Module). The bit corresponding to the value of this variable in the traversal map of a class will be set or cleared depending on whether the class has been examined on that traversal or not. **CTID** cannot equal or exceed the value of **MAX_TRAVERSALS** since this is the maximum number of bits in any class traversal map.

**INSTANCE_SLOT_REF**

PURPOSE: The string used to attach a direct slot reference to an instance-set member.

OTHER NOTES: Implemented as a preprocessor constant.

**QueryCore**

PURPOSE: A general state variable storing the test expression, distributed action expressions and the solution sets of instance addresses for the currently executing instance-set query function (see the general notes).

**QueryCoreStack**

PURPOSE: A pointer to a stack of instance-set query function "cores" (see **QueryCore**). The order of the stack indicates a nesting of instance-set query functions with the topmost core corresponding to the currently executing query (see the general notes).

**GLOBAL FUNCTIONS**

**AnyInstances**

PURPOSE: Determines if any instance-sets satisfy a query.

RETURNS: A non-zero integer if there were any instance-sets which satisfied the query, zero otherwise.

OTHER NOTES: Implementation of the CLIPS function **any-instancep**.

**Bit Access Functions**

PURPOSE: **testbit**, **setbit** and **clearbit** are used in parsing defclasses to check for qualifier duplication as well as in marking class
hierarchy traversals (see the general notes on the Class Commands Module).

OTHER NOTES: Implemented as a preprocessor macros in object.h.

**DelayedQueryDoForAllInstances**

PURPOSE: Performs an action for each instance-set which satisfies a query after determining all such sets.

ARGUMENTS: A data object buffer to hold the result of evaluating the action on the last instance-set which satisfied the query.

OTHER NOTES: Implementation of the CLIPS function `delayed-do-for-all-instances`.

**GetQueryInstance**

PURPOSE: References to instance-set member variables within a query are replaced with calls to this function (see `ReplaceInstanceVariables`). The first CLIPS supplied argument is the nesting depth of the applicable query function and is used to find the appropriate query core. The second argument is a positional index into the query core’s solution array and is used to find the appropriate instance in an instance-set satisfying a query.

ARGUMENTS: A data object buffer to hold the name of the instance to which the instance-set member variable refers.

OTHER NOTES: Implementation of the internal CLIPS function `(query-instance)`.

**GetQueryInstanceSlot**

PURPOSE: Direct slot references of instance-set member variables within a query are replaced with calls to this function (see `ReplaceSlotReference`). The first CLIPS supplied argument is the nesting depth of the applicable query function and is used to find the appropriate query core. The second argument is a positional index into the query core’s solution array and is used to find the appropriate instance in an instance-set satisfying a query. The third argument is the symbolic name expression of the slot.

ARGUMENTS: A data object buffer to hold the value of the direct slot reference of the instance-set member variable.
GetTraversalID

PURPOSE: Gets a new unused integer id for a class hierarchy traversal. The bit corresponding to the new id is cleared in all the existing class traversal maps (see the general notes in the Class Commands Module).

RETURNS: An integer indicating the new class hierarchy traversal id.

OTHER NOTES: The global variable CTID is used for the new id. An error will be generated if CTID already equals or exceeds MAX_TRAVERSALS, for this means that all the bits in class traversal maps are in use. Uses the clearbit function.

QueryDoForAllInstances

PURPOSE: Performs an action for each instance-set which satisfies a query as each set is determined.

ARGUMENTS: A data object buffer to hold the result of evaluating the action on the last instance-set which satisfied the query.

OTHER NOTES: Implementation of the CLIPS function do-for-all-instances.

QueryDoForInstance

PURPOSE: Performs an action for the first instance-set which satisfies a query.

ARGUMENTS: A data object buffer to hold the result of evaluating the action for the instance-set which satisfied the query.

OTHER NOTES: Implementation of the CLIPS function do-for-instance.

QueryFindAllInstances

PURPOSE: Groups all instance-sets which satisfy a query into a multifield variable.

ARGUMENTS: A data object buffer to hold the multifield result.

OTHER NOTES: Implementation of the CLIPS function find-all-instances.
**QueryFindInstance**

**PURPOSE:** Groups the first instance-set which satisfies a query into a multifield variable.

**ARGUMENTS:** A data object buffer to hold the multifield result.

**OTHER NOTES:** Implementation of the CLIPS function `find-instance`.

**ReleaseTraversalID**

**PURPOSE:** The last allocated class hierarchy traversal id is released for later reuse.

**OTHER NOTES:** the internal integer variable `CTID` is merely decremented.

**SetTraversalID**

**PURPOSE:** This function is used when recursively examining classes to mark which ones have already been visited. This is to avoid examining any classes more than once due to multiple inheritance.

**ARGUMENTS:**
1) A class's hierarchy traversal map.
2) The traversal id to mark as used.

**OTHER NOTES:** Implemented as a preprocessor macro in insquery.h.

The bit corresponding to the id is set in the class's traversal map. Uses the `setbit` function.

**SetupQuery**

**PURPOSE:** Support routine for `SetupObjectSystem` in the Class Commands Module which defines all functions and commands for instance-set queries.

**OTHER NOTES:** Initialization differs between standard and run-time configurations.

**TestTraversalID**

**PURPOSE:** This function is used when recursively examining classes to test if a particular class has already been visited. This is to avoid examining any classes more than once due to multiple inheritance.

**ARGUMENTS:**
1) A class's hierarchy traversal map.
2) The traversal id test.
OTHER NOTES: Implemented as a preprocessor macro in insquery.h. Uses the testbit function.

INTERNAL FUNCTIONS

AddSolution

PURPOSE: Support routine for TestEntireClass which takes the most recently found instance-set which satisfies the query (given by the "solutions" field of the query core) and adds it to a list of instance-sets which have all satisfied the query.

OTHER NOTES: This function is only called by TestEntireClass for query functions which require all the solutions to be grouped: QueryFindAllInstances and DelayedQueryDoForAllInstances.

The set of solutions is stored as a list of arrays of instance addresses. Each array in the list is an instance-set which satisfies the query, and each element in the array is the address of an instance that positionally matches the respective instance-set member variable. However, each array holds one extra element to be used as a pointer to the next instance-set.

DeleteQueryClasses

PURPOSE: Support routine for all the query functions which deallocates the list of lists of classes which form an instance-set template.

ARGUMENTS: A pointer to an instance-set template.

OTHER NOTES: The "busy" counts for the classes in the restriction lists are decremented.

DetermineQueryClasses

PURPOSE: Support routine for all the query functions which creates a series of query restriction classes to form an instance-set template.

ARGUMENTS: 1) A series of expressions representing the class restrictions for the instance-set members. Each class restriction list is separated by the special symbol expression QUERY_DELIMITER_SYMBOL. 2) The name of the calling function. 3) An integer buffer for the number of instance-set member
variables (each member of the set can have multiple class restrictions).

RETURNS: A pointer to the first instance-set template node, or NULL on errors.

---

### FindQueryCore

**PURPOSE:** Support routine for `GetQueryInstance` and `GetQueryInstanceSlot` which finds a particular query core in the stack of cores for nested instance-set query functions.

**ARGUMENTS:** An integer indicating the number instance-set query functions which nest the core of the one of interest.

**RETURNS:** A pointer to the appropriate query core.

**OTHER NOTES:** A nesting depth of 0 gets the core directly from the `QueryCore` variable. Greater depths access the variable `QueryCoreStack` in a top-down fashion.

---

### FormChain

**PURPOSE:** Support routine for `DetermineQueryClasses` which creates a new class pointer restriction node(s) to be added to a class restriction list in the instance-set template.

**ARGUMENTS:**
1) The name of the calling function.
2) A data object holding the symbolic name of a class or a multifield which fields are all symbolic names of classes.

**RETURNS:** A pointer to a instance-set template node, or NULL on errors.

**OTHER NOTES:** The "busy" counts for the class(es) in the restriction list is incremented.

---

### IsQueryFunction

**PURPOSE:** Support routine for `ReplaceInstanceVariables` which determines if an action in an instance-set query function is a call to another nested instance-set query function. If it is, then all instance-set member variable references in the nested query function will reference a core with an index one greater than the one currently being parsed.

**ARGUMENTS:** A pointer to an action expression.

**RETURNS:** A non-zero integer if the expression is a call to one of the following functions: `AnyInstances`, `QueryFindInstance`, `QueryFindAllInstances`, `QueryDoForInstance`,
**ParseQueryAction**

*Purpose:* Parses `do-for-instance`, `do-for-all-instances` and `delayed-do-for-all-instances` function calls into a series of expressions that can later be evaluated by `EvaluateExpression`.

*Arguments:* 1) An expression node containing the function call. 2) The logical name of the input source.

*Returns:* The top of series of expressions representing the function call, or NULL on errors.

*Other Notes:* This special function parser is required because these functions do not follow the standard format of CLIPS functions, e.g. instance-set template member variable's class restrictions would like function calls to the standard CLIPS function parser.

```
<function> (<instance-set member>+)
<query> <action>)
<instance-set member> ::= (<variable> <class>+)
```

is parsed to the following:

```plaintext
  Function
    ↓ ARG
  Query Expression NEXT-ARG Action Expression
    ↓ ARG
  Member Class 1 NEXT-ARG Member Class 2 NEXT-ARG ... QUERY DELIMIT STMBOL
    ↓ ARG
  Member Class 1 NEXT-ARG Member Class 2 NEXT-ARG ... QUERY DELIMIT STMBOL
    ↓ ARG
  ...
### ParseQueryActionExpression

**PURPOSE:** Support routine for `ParseQueryAction` which parses the distributed action for an instance-set query function.

**ARGUMENTS:**
1) A series of expressions that represent the parsed form of the instance-set query function so far.
2) The logical name of the input source.
3) A series of expressions (generated by `ParseQueryRestrictions`) listing the names of the instance-set member variables.

**RETURNS:** A non-zero integer if the action was parsed successfully, zero otherwise.

**OTHER NOTES:** A check is made to insure that no attempts are made in the action to rebind any instance-set member variables.

### ParseQueryNoAction

**PURPOSE:** Parses `any-instancep`, `find-instance` and `find-all-instances` function calls into a series of expressions that can later be evaluated by `EvaluateExpression`.

**ARGUMENTS:**
1) An expression node containing the function call.
2) The logical name of the input source.

**RETURNS:** The top of series of expressions representing the function call, or NULL on errors.

**OTHER NOTES:** This special function parser is required because these functions do not follow the standard format of CLIPS functions, e.g. instance-set template member variable's class restrictions would like function calls to the standard CLIPS function parser.

\[
\text{(<function> (<instance-set member>+)}<query>)
\text{(<instance-set member> ::= (<variable> <class>+)}
\]
ParseQueryRestrictions

Purpose: Support routine for ParseQueryAction and ParseQueryNoAction which instance-set template for a query function, i.e., instance-set member variables and their class restrictions.

Arguments: 1) A series of expressions that represent the parsed form of the instance-set query function so far. 2) The logical name of the input source. 3) A buffer to use for scanned tokens.

Returns: A series of expressions listing the names of the instance-set member variables.

Other Notes: A check is made to insure there are no duplicate instance-set member variables. In addition to generating the list for the return value, the class restriction lists are attached to the main expression (argument #1) as described in ParseQueryAction and ParseQueryNoAction.

ParseQueryTestExpression

Purpose: Support routine for ParseQueryAction and ParseQueryNoAction which parses the test expression for an instance-set query function.

Arguments: 1) A series of expressions that represent the parsed form of the instance-set query function so far. 2) The logical name of the input source.
RETURNS: A non-zero integer if the test expression was parsed successfully, zero otherwise.

OTHER NOTES: A check is made to insure that no binds occur in the test.

---

PopQueryCore

PURPOSE: Support routine for all the instance-set query functions which pops the first core of the query core stack (QueryCoreStack) and assigns it to the current query core (QueryCore).

OTHER NOTES: This routine is called at the end of every instance-set query function.

---

PushQueryCore

PURPOSE: Support routine for all the instance-set query functions which pushes the current query core (QueryCore) onto the query core stack (QueryCoreStack).

OTHER NOTES: This routine is called at the beginning of every instance-set query function.

---

ReplaceInstanceVariables

PURPOSE: Support routine for ParseQueryAction and ParseQueryNoAction which recursively replaces instance-set member variable references in the test and action expressions of an instance-set query function with appropriate calls to the internal functions GetQueryInstance and GetQueryInstanceSlot.

ARGUMENTS: 1) A series of expressions (generated by ParseQueryRestrictions) listing the names of the instance-set member variables. 2) The test or action expression in which to replace variable references. 3) An integer flag indicating whether to accept direct slot references (1) or not (0). 4) The number of instance-set query functions which nest the one currently being parsed.

OTHER NOTES: If a recursive call is made on an expression which is another instance-set query function call, that recursive call will be passed argument #4 plus one. This recursive "counting" mechanism will correspond one-to-one with the PushQueryCore and PopQueryCore calls when the instance-set query function is actually executed. This is why
argument #4 is a valid index for which query core to select at run-time from the stack.

### ReplaceSlotReference

**PURPOSE:** Support routine for ReplaceInstanceVariables which replaces direct slot references with calls to the internal function GetQueryInstanceSlot.

**ARGUMENTS:**
1. A series of expressions (generated by ParseQueryRestrictions) listing the names of the instance-set member variables.
2. The test or action expression in which to replace variable references.
3. The address of the CLIPS function (query-instance-slot).
4. The number of instance-set query functions which nest the one currently being parsed.

### TestEntireChain

**PURPOSE:** Support routine for QueryFindAllInstances, QueryDoForAllInstances and DelayedQueryDoForAllInstances which examines all the instances of classes (and their subclasses) in a class restriction list of a particular member variable in the instance-set template.

**ARGUMENTS:**
1. A pointer into the instance-set template indicating the class restriction list for the instance-set member variable being tested. The "next" link of the top node points to the top node of the next member variable's class restriction list, and the "chain" links form the class restriction list for this member variable.
2. The relative integer index of the instance-set member variable being tested.

**OTHER NOTES:** This function is mutually recursive with TestEntireClass.

The following is a synopsis of TestEntireChain:

1. Set the AbortQuery flag

2. For all classes in the class restriction list do:
   2a. Clear the AbortQuery flag.
   2b. Get a unique class traversal id.
   2c. Call TestEntireClass for the class.
   2d. Release the traversal id.
2e. Abort if this member variable or any after it did not have any instances which satisfied the query.

**TestEntireClass**

**PURPOSE:** Support routine for **TestEntireChain** which examines all the instances of a class and its subclasses.

**ARGUMENTS:**
1) A class hierarchy traversal id to use when recursively examining subclasses.
2) A pointer to the class.
3) A pointer into the instance-set template indicating the class restriction list for the instance-set member variable being tested.
4) The relative integer index of the instance-set member variable being tested.

**OTHER NOTES:** This function is self-recursive and mutually recursive with **TestEntireChain**. The self-recursion is to test subclasses. **TestEntireClass** calls **TestEntireChain** until it reaches the last class restriction list. In this manner, all permutations are examined, varying the the instances matching the last member variable first (as described in the *Basic Programming Guide*).

The following is a synopsis of **TestEntireClass**:

1. If this class has already been examined for this traversal id, immediately return. Otherwise, set the traversal id bit for this class.

2. Save and set the **MaintainGarbageInstances** flag (see the Instance Functions Module) to insure that instance links can be followed even in the event an instance is deleted as the result of an instance-set query or action.

3. For every instance of this class do:
   3a. Place this instance in the appropriate position in the solution corresponding to the instance-set member variable being tested.
   3b. If there are no instance-set member variables remaining to be tested on this pass through the template, go to 3d. Otherwise, call **TestEntireChain** for the class restriction list of the next instance-set member variable. Go to the beginning of 3.
   3d. A complete instance-set has been generated. Evaluate the query expression. If it is satisfied, either evaluate the query action or add the instance-set to the list of solutions (see **AddSolution**), depending on which instance-set
query function is being executed.

4. Restore the **MaintainGarbageInstances** flag.

5. Call **TestEntireClass** for every subclass of the current class.

---

**TestForFirstInChain**

**PURPOSE:** Support routine for **AnyInstances**, **QueryFindInstance** and **QueryDoForInstance** which examines all the instances of classes (and their subclasses) in a class restriction list of a particular member variable in the instance-set template.

**ARGUMENTS:**

1) A pointer into the instance-set template indicating the class restriction list for the instance-set member variable being tested. The "next" link of the top node points to the top node of the next member variable's class restriction list, and the "chain" links form the class restriction list for this member variable.

2) The relative integer index of the instance-set member variable being tested.

**RETURNS:** A non-zero integer if an instance was found for the instance-set member variable which was part of an instance-set which satisfied the query.

**OTHER NOTES:** This function is mutually recursive with the **TestForFirstInstanceInClass**.

The following is a synopsis of **TestForFirstInChain**:

1. Set the **AbortQuery** flag

2. For all classes in the class restriction list do:
   2a. Clear the **AbortQuery** flag.
   2b. Get a unique class traversal id.
   2c. Call **TestForFirstInstanceInClass**, and, if it indicates success, release the traversal id and return success.
   2d. Release the traversal id.
   2e. Return failure if this member variable or any after it did not have any instances which satisfied the query.

3. Return failure.
TestForFirstInstanceInClass

PURPOSE: Support routine for TestForFirstInChain which examines all the instances of a class and its subclasses.

ARGUMENTS:
1) A class hierarchy traversal id to use when recursively examining subclasses.
2) A pointer to the class.
3) A pointer into the instance-set template indicating the class restriction list for the instance-set member variable being tested.
4) The relative integer index of the instance-set member variable being tested.

RETURNS: A non-zero integer if an instance of the class was part of an instance-set which satisfied the query.

OTHER NOTES: This function is self-recursive and mutually recursive with TestForFirstInChain. The self-recursion is to test subclasses. TestForFirstInstanceInClass calls TestForFirstInChain until it reaches the last class restriction list. In this manner, all permutations are examined, varying the the instances matching the last member variable first (as described in the Basic Programming Guide).

The following is a synopsis of TestForFirstInstanceInClass:

1. If this class has already been examined for this traversal id, immediately return. Otherwise, set the traversal id bit for this class.

2. Save and set the MaintainGarbageInstances flag (see the Instance Functions Module) to insure that instance links can be followed even in the event an instance is deleted as the result of an instance-set query or action.

3. For every instance of this class do:
   3a. Place this instance in the appropriate position in the solution corresponding to the instance-set member variable being tested.
   3b. If there are no instance-set member variables remaining to be tested on this pass through the template, go to 3d. Otherwise, call TestForFirstInChain for the class restriction list of the next instance-set member variable, and, if success is returned, go to 4. Go to the beginning of 3.
   3d. A complete instance-set has been generated. Evaluate the query expression. If it is satisfied, either evaluate the query action or do nothing, depending on which instance-set query function is being executed.
4. Restore the `MaintainGarbageInstances` flag.

5. Return success if an instance-set which satisfied the query was successfully completed within the step 3 loop.

6. Call `TestForFirstInClass` for every subclass of the current class and return success immediately if any return success.

7. Return failure.
The Definstances Module (defins.c) provides the capability needed to implement the definstances construct. For a description of the definstances construct, see the Basic Programming Guide. The definstances construct capability, along with the other features of the CLIPS Object-Oriented Language (COOL), can be removed by using the appropriate compile flag in the setup header file. The definstances data structure is summarized in the following diagram:

<table>
<thead>
<tr>
<th>Name (Symbol Pointer)</th>
<th>Busy Count (int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make-Instance Call (Expression Pointer)</td>
<td>Binary Load/Save Index (long int)</td>
</tr>
<tr>
<td>Pretty-Print Form (array of char)</td>
<td>Previous Link (Definstances Pointer)</td>
</tr>
<tr>
<td>Next Link (Definstances Pointer)</td>
<td></td>
</tr>
</tbody>
</table>

The internal data structure of a definstances consists of a symbolic name and a series of expressions forming a call to `make-instance`. A non-zero busy count for a definstances indicates that instances in that definstances are currently being created, and it is not safe to delete the definstances. Other fields in the structure include: the pretty-print form, an index for use in binary load/save and the construct compiler and pointers for double links to other definstances.

**GLOBAL VARIABLES**

<table>
<thead>
<tr>
<th>DefinstancesList</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: A pointer to the first node in the list of all currently defined definstances.</td>
</tr>
</tbody>
</table>

**INTERNAL VARIABLES**

<table>
<thead>
<tr>
<th>DefinstancesListBottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: A pointer to the first node in the list of all currently defined definstances.</td>
</tr>
</tbody>
</table>

**GLOBAL FUNCTIONS**

<table>
<thead>
<tr>
<th>ClearDefinstances</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE: Used by the clear command to remove all currently defined definstances.</td>
</tr>
</tbody>
</table>
RETURNS: The integer zero if not all definstances were successfully cleared, non-zero otherwise.

**CmdListDefinstances**

PURPOSE: Lists all the currently defined definstances.

OTHER NOTES: Implementation of the CLIPS function `list-definstances`.

**CmdUndefinstances**

PURPOSE: Removes a definstances.

OTHER NOTES: Implementation of the CLIPS function `undefinstances`.

**Embedded Access for Definstances**

PURPOSE: The following functions are provided for embedded access and are documented in the *Advanced Programming Guide*: `DeleteDefinstances`, `FindDefinstances`, `GetDefinstancesName`, `GetDefinstancesPPForm`, `GetNextDefinstances`, `IsDefinstancesDeletable` and `ListDefinstances`.

**PPDefinstances**

PURPOSE: Displays the pretty-print form of the definstances specified by the CLIPS supplied argument.

OTHER NOTES: Implementation of the CLIPS function `ppdefinstances`.

**SetDefinstancesList**

PURPOSE: Initializes the global variables `DefinstancesList` and `DefinstancesListBottom` to point to the top and bottom respectively of the given list of definstances.

ARGUMENTS: A pointer to the top of a list of definstances.

OTHER NOTES: This function is used only in a run-time version of CLIPS.

**SetupDefinstances**

PURPOSE: Support routine for `SetupObjectSystem` in the Class Commands Module which defines all functions and commands for the definstances construct. Sets up all necessary `load`, `clear`, `save` and `reset` interfaces.
OTHER NOTES: Initialization differs between standard and run-time configurations.

INTERNAL FUNCTIONS

**FindDefinstancesBySymbol**

PURPOSE: Determines the address of a specified definstances.

ARGUMENTS: A pointer to a symbol.

RETURNS: A pointer to a definstances.

**InitializeDefinstances**

PURPOSE: Used by the *reset* command to delete all existing instances of user-defined classes (via *delete* messages) and create the ones in definstances (via *make-instance* calls).

**ParseDefinstances**

PURPOSE: Used by the *load* command to parse a definstances.

ARGUMENTS: The logical name of the input source.

RETURNS: The integer zero if there are no parsing errors, non-zero otherwise.

**ParseDefinstancesName**

PURPOSE: Support routine for *ParseDefinstances* which parses the definstances name and an optional comment.

ARGUMENTS: The logical name of the input source.

RETURNS: The symbolic name of the definstances, NULL on errors.

**RemoveDefinstances**

PURPOSE: Removes a definstances.

ARGUMENTS: A pointer to a definstances.

RETURNS: A no-zero integer if the definstances was successfully deleted, zero otherwise.
SaveDefinstances

PURPOSE: Used by the save command to write out the pretty-print forms of all the currently defined definstances.

ARGUMENTS: The logical name of the output destination.
The Object Construct Compiler Interface (objcmp.c) Module provides the interface for COOL to the *constructs-to-c* command.
Object Binary Load/Save Interface Module

The Object Binary Load/Save Interface (objbin.c) Module provides the interface for COOL to the `bload/bsave` commands.
Main Module

The Main Module (main.c) contains the only functions which should have to be modified to add extensions or embed CLIPS under normal circumstances.

GLOBAL VARIABLES

None.

LOCAL VARIABLES

None.

GLOBAL FUNCTIONS

main

PURPOSE: Startup function for CLIPS. Under normal operation, this function initializes CLIPS, checks for command line arguments, then calls the CommandLoop function. See the Advanced Programming Guide for details on embedding CLIPS.

UserFunctions

PURPOSE: Called during CLIPS initialization. Allows users to insert their own function definition calls. See the Advanced Programming Guide for details on integrating CLIPS.

INTERNAL FUNCTIONS

None.
### Index

<table>
<thead>
<tr>
<th>AddAbortBloadFunction</th>
<th>245</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddActivation</td>
<td>157, 163</td>
</tr>
<tr>
<td>AddAfterBloadFunction</td>
<td>245</td>
</tr>
<tr>
<td>AddBeforeBloadFunction</td>
<td>246</td>
</tr>
<tr>
<td>AddBinaryItem</td>
<td>98, 107, 204, 214, 234, 238</td>
</tr>
<tr>
<td>AddBindName</td>
<td>50</td>
</tr>
<tr>
<td>AddBloadFunctionToList</td>
<td>247</td>
</tr>
<tr>
<td>AddBloadReadyFunction</td>
<td>246, 247</td>
</tr>
<tr>
<td>AddBreakpoint</td>
<td>163</td>
</tr>
<tr>
<td>AddClass</td>
<td>334, 339, 340, 341, 342, 371, 383</td>
</tr>
<tr>
<td>AddCleanupFunction</td>
<td>77, 84</td>
</tr>
<tr>
<td>AddClearBloadReadyFunction</td>
<td>246</td>
</tr>
<tr>
<td>AddClearFunction</td>
<td>70, 285, 303, 327</td>
</tr>
<tr>
<td>AddCodeGeneratorItem</td>
<td>98, 107, 204, 214, 252, 254</td>
</tr>
<tr>
<td>AddConstruct</td>
<td>70</td>
</tr>
<tr>
<td>AddCPFFunction</td>
<td>84</td>
</tr>
<tr>
<td>AddDeffunction</td>
<td>284</td>
</tr>
<tr>
<td>AddDefglobal</td>
<td>107</td>
</tr>
<tr>
<td>AddDefrule</td>
<td>199</td>
</tr>
<tr>
<td>AddDeftemplate</td>
<td>208</td>
</tr>
<tr>
<td>AddDouble</td>
<td>20, 249</td>
</tr>
<tr>
<td>AddEphemeralFloat</td>
<td>25</td>
</tr>
<tr>
<td>AddEphemeralInteger</td>
<td>25</td>
</tr>
<tr>
<td>AddEphemeralSymbol</td>
<td>25</td>
</tr>
<tr>
<td>AddFact</td>
<td>86, 87, 191</td>
</tr>
<tr>
<td>AddFunctionParser</td>
<td>42</td>
</tr>
<tr>
<td>AddGeneric</td>
<td>301, 308</td>
</tr>
<tr>
<td>AddGenericMethod</td>
<td>306</td>
</tr>
<tr>
<td>AddHashDeftemplate</td>
<td>213</td>
</tr>
<tr>
<td>AddHashedFact</td>
<td>86</td>
</tr>
<tr>
<td>AddHashFunction</td>
<td>42</td>
</tr>
<tr>
<td>AddLogicalDependencies</td>
<td>191, 193</td>
</tr>
<tr>
<td>AddLong</td>
<td>20, 249</td>
</tr>
<tr>
<td>AddMethod</td>
<td>302, 306</td>
</tr>
<tr>
<td>AddParameter</td>
<td>294</td>
</tr>
<tr>
<td>AddPeriodicFunction</td>
<td>77, 84</td>
</tr>
<tr>
<td>AddResetFunction</td>
<td>70</td>
</tr>
</tbody>
</table>

| dfs-getbind           | 283 |
| dfs-runknown          | 283 |
| dfs-wildargs          | 283 |
| gnrc-bind             | 292 |
| gnrc-runknown         | 293 |
| gnrc-wildargs         | 292 |
| hndgetbind            | 373 |
| hndunknown            | 375 |
| hndwildargs           | 375 |
| (query-instance-slot) | 400, 408 |
| (query-instance)      | 399 |
| * 51, 263             | |
| ** 273                | |
| + 51, 263             | |
| - 51, 263             | |
| / 51, 263             | |
| < 51, 263             | |
| <= 51, 263            | |
| <> 51, 263            | |
| = 51, 263             | |
| > 51, 263             | |
| >= 51, 263            | |
| Abort                 | 28, 29 |
| AbortBload            | 247 |
| AbortBloadFunctions   | 244, 245 |
| AbortExit             | 29 |
| AbortQuery            | 397, 408, 410 |
| abs                   | 267 |
| ABSTRACT_RLN          | 319 |
| ACCESS_BIT            | 319 |
| acos                  | 273 |
| acosh                 | 273 |
| acot                  | 273 |
| acoth                 | 273 |
| acsc                  | 273 |
| acsch                 | 273 |
| ActivateRouter        | 29 |
| ActivationBasis       | 162 |
| active instance        | 347, 362, 371, 382 |
| ActualPoolSize        | 9 |
AddressesToStrings 76
AddRouter 29
AddRunFunction 163
AddSaveFunction 71
AddSingleMatch 181
AddSolution 402, 409
AddSymbol 18, 20, 250
AddSystemClass 338, 339
AddSystemHandlers 371, 376
AddTerminatorJoin 199
AddToList 339
AddToDependencyList 191
AddToSegmentList 87
AddWatchItem 78
AdjacentReduction 119
AdjustFieldPosition 182
AfterBloadFunctions 244, 245
Agenda 161, 162, 163, 164, 166, 167, 168, 169, 170, 205
AgendaChanged 161, 164, 167
AllocateBlock 15
AllocateChunk 16
AllVariablesInPattern 132
ALL_QUALIFIER 347
analysis.c 2, 127
AnalysisExpressions 121, 122, 123, 124, 125
and 51, 136, 158, 263
and CE 109, 113, 114, 117
any-instancep 398, 405
AnyInstances 398, 410
AppendCommandString 65
AppendNToString 78
AppendStrings 39
AppendToString 78
ArgumentParse 42, 43
asec 273
asech 273
asin 273
asinh 273
assert 2, 3, 53, 54, 93, 217, 220
AssertParse 50
AssertRetractInProgress 85
AssertSlotsMultiplyDefined 220
AssertString 87
AssignmentParse 113
atan 273
atanh 273
AtomDeinstall 78
AtomInstall 79
AutoFloatDividend 263
batch 277
BDefglobalArray 101
BDefglobalPointersArray 101
BeforeBloadFunctions 244, 245, 246
BeforeClearFunction 69, 71, 73
BeforeResetFunction 69, 73
BEGIN_TRACE 280, 301, 381
BIG_PRIME 334, 359
BinaryFileHandle 244, 247, 248, 249
BinaryFP 245, 247, 248, 249
BinaryPrefixID 241, 243
BinaryRefNum 245, 247, 248
BinaryVersionID 241, 243
bind 261
bind list 280, 282, 283, 287, 290, 292, 297, 302, 304, 370, 373, 378, 382, 388, 391
BindList 57, 59
BindParse 50
Bit Access Functions 398
BITS_PER_BYTE 397
bload 3, 95, 97, 98, 101, 104, 107, 203, 204, 207, 208, 211, 214, 238, 239, 243, 244, 245, 246, 247, 249, 250, 277, 280, 285, 313, 419
bload.c 3, 243
BloadActive 245, 247
BloadCommand 246
Bloaded 247
BloadExpressions 247
BloadReadyFunctions 245, 246
BlockInfoSize 8
BlockMemoryInitialized 8
break 261, 282, 304, 391
BreakContext 261
BreakFlag 261  
browse-classes 320  
BrowseClass 322, 327  
BrowseClassesCmd 320  
bsave.c 3, 233, 243  
BsaveAllExpressions 240  
BsaveCommand 239  
BsaveExpression 239, 240  
build 271  
build.c 2, 143  
BuildDefaultSlots 366  
BuildInstance 360, 366, 367  
BuildNetworkExpressions 132  
BuildPartialOrders 339, 342  
BuildRHSAssert 53, 98  
BuildSubclassLinks 339  
call-next-handler 381, 387  
call-next-method 303, 388  
CallClearFunctions 71  
CallDeffunction 280, 281, 285  
CallHandlers 382, 384, 388, 389, 391, 392  
callNextHandler 381, 384, 388, 389, 391  
callNextMethod 302  
CARDINALITY_BIT 319  
CatchControlC 5  
ChangesToInstances 357  
ChangeToFactList 85, 89, 92  
ChangeToGlobals 101, 103, 106  
CheckArgListParse 51  
CheckClass 324  
CheckClassAndSlot 324  
CheckCurrentMessage 382  
CheckDeffunctionCall 285  
CheckExpression 130  
CheckFactAddress 132  
CheckForPrimableJoins 199  
CheckGenericExists 294  
CheckHandlerAgainstSlots 371  
CheckHandlerArgCount 389  
CheckHandlerBindList 376  
CheckInstanceAndSlot 353  
CheckLHSSlotTypes 230  
CheckMethodExists 294  
CheckMultifieldSlotInstance 353  
CheckMultifieldSlotModify 358, 360, 362  
CheckPattern 132  
CheckRHSSlotTypes 220  
CheckSlotAllowedValues 208, 218  
CheckSlotConflicts 225  
CheckSlotRange 208, 218  
CheckSlotType 209, 218  
CheckTemplateFact 218  
CheckTwoClasses 324  
CheckVariables 131  
ChunkInfoSize 8  
class 293, 315, 329, 348  
class-abstractp 322  
class-existp 322  
class-message-handler-existp 320  
class-message-handlers 320  
class-slot-existp 320  
class-slots 321  
class-subclasses 321  
class-superclasses 321  
class precedence list 330, 336  
classcom.c 4, 315  
ClassExistError 334  
classfun.c 4, 329  
classfun.h 333  
ClassHandlersCmd 320, 325  
ClassHasHandler 320  
ClassHasSlot 320, 324  
ClassInfoFnxArgs 325  
ClassList 313, 317, 332, 333, 338  
ClassListBottom 334, 338  
ClassSlotsCmd 321, 325  
ClassSubclassesCmd 325  
ClassSuperclassesCmd 321, 325  
ClassTable 317, 333  
CLASS_RLN 347  
CLASS_TABLE_HASH_SIZE 333  
CleanupInstances 361, 367
<table>
<thead>
<tr>
<th>Function/Method/Class</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>92, 97, 104, 198, 209, 211, 238, 245, 261, 285, 303, 327, 413</td>
</tr>
<tr>
<td>clearbit</td>
<td>398, 400</td>
</tr>
<tr>
<td>ClearBload</td>
<td>248</td>
</tr>
<tr>
<td>ClearBloadReadyFunctions</td>
<td>245, 246</td>
</tr>
<tr>
<td>ClearCLIPS</td>
<td>71</td>
</tr>
<tr>
<td>ClearDefclasses</td>
<td>335, 340, 383, 386</td>
</tr>
<tr>
<td>ClearDeffacts</td>
<td>98</td>
</tr>
<tr>
<td>ClearDeffunctions</td>
<td>285</td>
</tr>
<tr>
<td>ClearDefgenerics</td>
<td>303</td>
</tr>
<tr>
<td>ClearDefglobals</td>
<td>102</td>
</tr>
<tr>
<td>ClearDefinstances</td>
<td>413</td>
</tr>
<tr>
<td>ClearDefmethods</td>
<td>303</td>
</tr>
<tr>
<td>ClearDefrules</td>
<td>195</td>
</tr>
<tr>
<td>ClearDeftemplates</td>
<td>209</td>
</tr>
<tr>
<td>ClearLowerBetaMemory</td>
<td>159</td>
</tr>
<tr>
<td>ClearParsedBindNames</td>
<td>49</td>
</tr>
<tr>
<td>ClearPatternMatches</td>
<td>182</td>
</tr>
<tr>
<td>ClearRuleFromAgenda</td>
<td>163</td>
</tr>
<tr>
<td>CLIPSDeleteInstance</td>
<td>348</td>
</tr>
<tr>
<td>CLIPSFalseSymbol</td>
<td>18, 23, 24</td>
</tr>
<tr>
<td>CLIPSFunctionCall</td>
<td>58</td>
</tr>
<tr>
<td>CLIPSGetSlot</td>
<td>348, 353</td>
</tr>
<tr>
<td>CLIPSInputCount</td>
<td>27</td>
</tr>
<tr>
<td>CLIPSMakeInstance</td>
<td>348, 353</td>
</tr>
<tr>
<td>CLIPSPutSlot</td>
<td>348, 353</td>
</tr>
<tr>
<td>CLIPSSendMessage</td>
<td>385, 391, 392</td>
</tr>
<tr>
<td>CLIPSSystemError</td>
<td>79</td>
</tr>
<tr>
<td>CLIPSTestSlot</td>
<td>348, 353</td>
</tr>
<tr>
<td>CLIPSTrueSymbol</td>
<td>18, 23, 24</td>
</tr>
<tr>
<td>CLIPSUnmakeInstance</td>
<td>348</td>
</tr>
<tr>
<td>close</td>
<td>75, 265</td>
</tr>
<tr>
<td>CloseAllFiles</td>
<td>29</td>
</tr>
<tr>
<td>CloseFile</td>
<td>29</td>
</tr>
<tr>
<td>CloseStringDestination</td>
<td>29</td>
</tr>
<tr>
<td>CloseStringSource</td>
<td>29</td>
</tr>
<tr>
<td>CmdListDefclasses</td>
<td>321</td>
</tr>
<tr>
<td>CmdListDeffunctions</td>
<td>282</td>
</tr>
<tr>
<td>CmdListDefgenerics</td>
<td>291</td>
</tr>
<tr>
<td>CmdListDefinstances</td>
<td>414</td>
</tr>
<tr>
<td>CmdListDefmessageHandlers</td>
<td>372</td>
</tr>
<tr>
<td>CmdListDefmethods</td>
<td>291</td>
</tr>
<tr>
<td>CmdListInstances</td>
<td>347</td>
</tr>
<tr>
<td>CmdPPDeffunction</td>
<td>282</td>
</tr>
<tr>
<td>CmdUndefclass</td>
<td>321, 326</td>
</tr>
<tr>
<td>CmdUndeffunction</td>
<td>282</td>
</tr>
<tr>
<td>CmdUndefgeneric</td>
<td>291</td>
</tr>
<tr>
<td>CmdUndefinstances</td>
<td>414</td>
</tr>
<tr>
<td>CmdUndefmessageHandler</td>
<td>372</td>
</tr>
<tr>
<td>CmdUndefmethod</td>
<td>291</td>
</tr>
<tr>
<td>CollectArguments</td>
<td>43</td>
</tr>
<tr>
<td>CombineExpressions</td>
<td>137</td>
</tr>
<tr>
<td>CommandLoop</td>
<td>65, 421</td>
</tr>
<tr>
<td>CommandString</td>
<td>64, 65, 66, 67</td>
</tr>
<tr>
<td>conmline.c 2</td>
<td>63</td>
</tr>
<tr>
<td>CompactActions</td>
<td>53</td>
</tr>
<tr>
<td>CompareBindings</td>
<td>168</td>
</tr>
<tr>
<td>CompleteCommand</td>
<td>63, 65</td>
</tr>
<tr>
<td>COMPOSITE_BIT</td>
<td>319</td>
</tr>
<tr>
<td>CONCRETE_RLN</td>
<td>319</td>
</tr>
<tr>
<td>ConjunctiveRestrictionParse</td>
<td>113</td>
</tr>
<tr>
<td>ConnectedPatternParse</td>
<td>113</td>
</tr>
<tr>
<td>conscomp.c 251</td>
<td>251</td>
</tr>
<tr>
<td>conserve-mem</td>
<td>267</td>
</tr>
<tr>
<td>ConserveMemory</td>
<td>8, 11, 14</td>
</tr>
<tr>
<td>constant 135, 136, 261</td>
<td>261</td>
</tr>
<tr>
<td>constant.h 298, 307, 313, 316, 332, 333, 338</td>
<td></td>
</tr>
<tr>
<td>ConstantExpression</td>
<td>43</td>
</tr>
<tr>
<td>constrct.c 2, 3, 41, 69</td>
<td>69</td>
</tr>
<tr>
<td>ConstructJoins</td>
<td>145</td>
</tr>
<tr>
<td>constructs-to-c</td>
<td>285, 311, 417</td>
</tr>
<tr>
<td>constructs-to-c 3, 97, 98, 104, 107, 203, 204, 207, 211, 214, 251, 254, 257</td>
<td></td>
</tr>
<tr>
<td>ConstructsToCCommand</td>
<td>254</td>
</tr>
<tr>
<td>ConstructsToCCommandDefinition</td>
<td>254</td>
</tr>
<tr>
<td>CopyClassLinks</td>
<td>339</td>
</tr>
<tr>
<td>CopyExpression</td>
<td>43</td>
</tr>
<tr>
<td>CopyMemory</td>
<td>9</td>
</tr>
<tr>
<td>CopyNodes</td>
<td>118</td>
</tr>
<tr>
<td>CopyPartialMatch</td>
<td>182</td>
</tr>
<tr>
<td>CopyPPBuffer</td>
<td>36</td>
</tr>
<tr>
<td>CopySegmentMarkers</td>
<td>175</td>
</tr>
<tr>
<td>CopyToken</td>
<td>36</td>
</tr>
<tr>
<td>CoreInitializeInstance</td>
<td>361, 362, 363, 364, 366</td>
</tr>
</tbody>
</table>
cos 273
cosh 273
cot 273
coth 273
CountArguments 43
CountJoins 121
CountPatternFields 121
CountSubclasses 325
CreateFact 87
CreateInitialFactDeffacts 95
CreateInitialPattern 113
CreateMultifield 87
CreateRawInstance 348, 360
CreateReadStringSource 33
CreateSystemClasses 338, 339, 340
crsv-trace-off 277
crsv-trace-on 277
csc 273
csch 273
CTID 397, 398, 400, 401
CurrentCore 381, 382, 388
CurrentDeffunctionName 280, 282
CurrentEphemeralCountMax 76
CurrentEphemeralSizeMax 76
CurrentEvaluationDepth 57
CurrentExpression 58
CurrentGeneric 299, 304
CurrentInstance 359, 366
CurrentMessageFrame 370, 379, 389
CurrentMessageName 379, 391
CurrentMessageSize 379
CurrentMethod 299
CurrentPatternFact 173, 174
CurrentPatternInfo 121, 125
CurrentPatternMarks 173, 174
CurrentTimetag 161
DeactivateRouter 30
DeallocateMarkedHandlers 377, 383, 387
DeclarationParse 114
DecrementFactCount 87
DecrementFloatCount 20
DecrementIndentDepth 37
DecrementInstanceCount 362
DecrementInstanceDepth 361
DecrementIntegerCount 21
DecrementSymbolCount 21
DefaultGetNextEvent 67
DefaultOutOfMemoryFunction 9
defclass 315, 329
Defclass Constants 319
Defclass Keywords 319
deffacts 54
deffacts.c 2, 95
deffactsArray 95
deffnctn.c 3, 279
deffunction 261, 279
Deffunction Bload/Bsave Functions 285
Deffunction Constructs-To-C Functions 285
deffunction parameter array 280, 281, 282, 286
Deffunction Trace Strings 280
deffunctionArray 280
DeffunctionError 280
DeffunctionGetBind 280, 282
defgeneric 289
DefglobalArray 101, 102, 105, 107
DefglobalPointersArray 101
DefglobalsArray 103
defglobl.c 2, 101
DefinedSlots 225, 227
DefineFunction 42, 45, 58, 90, 93, 205, 251, 255, 257, 258, 261, 263, 265, 267, 269, 271, 273, 275, 277
defins.c 4, 413
definstances 413
DefinstancesList 413, 414
DefinstancesListBottom 413, 414
defmessage-handler 369, 379
defmethod 289
defrule.c 2, 195
DefruleArray 203
DefruleBinarySetup 204
DefruleCommands 198, 205
DefruleHasBreakpoint 163
ExpressionCount 238, 239, 240, 253
ExpressionDeinstall 44
ExpressionFP 253, 255, 256
ExpressionHeader 253
ExpressionInstall 44
ExpressionSize 44
ExpressionToCode 252, 254
ExpressionVersion 253
expressn.c 1, 41
ExtractAnds 133
facet
    composite 342
    default 318
default-dynamic 318
    initialize-only 318, 358, 359
    local 318, 346
    multiple 318
    no-inherit 334, 341, 343
    read-only 318, 358
    read-write 318
    shared 318, 346, 366
    single 318
fact-index 93
factcom.c 2, 93
FactCompare 88
FactDeinstall 88, 91
FactDuplication 85, 89, 91
FactExists 88
FactHashTable 85, 86, 87, 88, 90, 92
FactInstall 88
FactList 85, 86, 89, 90, 91, 98
factmnr.c 2, 85
facts 2, 93
FalseSymbol 18, 19
FastFindFunction 248, 249
FastLoadFilePointer 32
FastLoadFilePtr 28, 31, 32, 33
FastSaveFilePointer 32
FastSaveFilePtr 28, 31, 32, 33
fetch 275
FieldCheckTemplate 221
FieldConversion 133
filecom.c 3

FileFunctionDefinitions 277
FilePrefix 253
find-all-instances 396, 400, 405
find-instance 401, 405
FindApplicableHandlers 379, 384, 386, 389, 390, 391, 392
FindApplicableMethods 304, 307, 308
FindApplicableOfName 385, 390
FindClassSlot 335, 386
FindDefclass 322
FindDefclassBySymbol 336
FindDeffacts 96
FindDeffunction 284
FindDeffunctionBySymbol 284
FindDefgeneric 291
FindDefgenericBySymbol 303
FindDefglobal 102
FindDefinstances 414
FindDefinstancesBySymbol 415
FindDefmessageHandler 372
FindDefrule 196
FindDeftemplate 209
FindFactInPartialMatch 182
FindFile 30
FindFptr 30
FindFunction 45, 142, 248
FindHandler 385
FindHandlerNameGroup 385
FindHandlerParameter 376
FindIndexedFact 88
FindInstance 348
FindInstance 336
FactCompare 88
FactExists 88
FactInstall 88
FieldCheckTemplate 221
FieldConversion 133
filecom.c 3
JSC-25014

FindQueryCore 403
FindSlot 210
FindSlotItem 210
FindSlotPosition 210
FindSymbol 21
FindSymbolMatches 21, 24
FindVariable 122
float 267
FloatArray 244
floatp 263
FloatsToCode 256
FloatTable 18, 19, 20, 21, 23, 24, 25, 26,
   39, 240, 241, 249, 255, 256, 257, 258, 259
FloatToString 80
FlushAlphaBetaMemory 183
FlushAnalysisExpressions 123
FlushCommandString 65
FlushGarbagePartialMatches 179
FlushPPBuffer 37
FlushSegments 89
FlushVariableAnalysis 123
ForceLogicalRetractions 191
format 265
FormChain 403
FormInstanceTemplate 329, 340, 341
free 10
Function0Parse 44
Function1Parse 45
Function2Parse 45
FunctionArray 244
FunctionBinarySize 240
FunctionHashTable 42, 46, 48
FunctionsToCode 257
garbage collection 345, 357, 358, 359,
   361, 365, 367
GarbageAlphaMatches 178, 179
GarbageFacts 86, 91
GarbagePartialMatches 178, 179, 183
GCALL 290
GDDCommand 211
genalloc 7, 10, 11, 12, 13
GenAnd 138
GenClose 248
GenConstant 138
generate.c 2, 135
GenerateCode 254, 256, 257, 258, 259
generic function 289, 299
Generic Function Trace Codes 300
Generic Function Trace Strings 301
GenericDispatch 290, 299, 304, 307
GenericInputToken 291
GenericList 290, 300, 306
GenericListBottom 300, 306
GenericStackTrace 300, 307
GenericStackSize 300
genexit 5
GenFourIntegers 138
genfree 7, 10, 11, 13
GenGetfield 138
GenGetvar 138
GenGetvarValue 138
GenJNColon 139
GenJNConstant 139
GenJNEq 139
GenJNVariableComparison 140
genlongalloc 10, 12
genlongfree 11, 14
genmemcpy 9, 11
GenOpen 248
GenOr 140
GenPNColon 140
GenPNConstant 140
GenPNEq 141
GenPNVariableComparison 141
genrand 5
genrobin.c 313
genrccmp.c 3, 311
genrccom.c 3, 289
genrcfun.c 3, 299
GenRead 247
genrealloc 11
genseed 5
gensym 267
gensym* 267
GensymNumber 267

CLIPS Architecture Manual 431
GetIntegerTable 22
GetJoinLogic 123
GetLHSSlots 230
GetLogicalName 81
GetMultiSlotPosition 218
GetNextActivation 164
GetNextDefclass 322
GetNextDeffacts 96
GetNextDeffunction 284
GetNextDefgeneric 291
GetNextDefglobal 102, 104
GetNextDefinstances 414
GetNextDefmessageHandler 372
GetNextDefmethod 292
GetNextDefrule 197
GetNextDeftemplate 211
GetNextFact 89
GetNextInstance 348
GetNextInstanceInClass 348
GetNextLogicalRetraction 191, 192
GetNextSymbolMatch 22
GetNode 118
GetNodeType 123
GetNotJoinExpression 124
GetNthWatchName 81
GetNthWatchValue 81
GetNumberOfActivations 164
GetNumberOfDefglobals 104
GetNumberOfFacts 89
GetNumericArgument 184
GetParsedBindNames 49
GetPatternExpression 124
GetPPBuffer 37
GetPPBufferStatus 37
GetPrimaryJoinExpression 124
GetPrintWhileLoading 71
GetQueryInstance 399, 403, 407
GetQueryInstanceSlot 399, 403, 407, 408
GetRelationForPattern 125
GetResetGlobals 104
GetResetGlobalsCommand 104
GetRHSPattern 54
GetRuleDeletions 197
GetRuleFiring 165
GetRulesWatch 197
GetSalienceEvaluation 165
GetSingleLHSSlot 230
GetSingleLHSSlots 231
GetSlotAssertValues 221
GetSlotFacets 322
GetSlotSources 322
GetStrategy 165
GetSymbolTable 22
GetToken 37
GetTraversalID 316, 397, 400
getvar 135
GetVariableDefinition 107, 108
GetVariableInformation 125
GetVariables 133
GetvarReplace 142
GetWatchItem 81
get_bind 261
get_end 261
get_field 136, 261
get_struct 11
get_var 181
get_var_struct 12
GlobalLHSBinds 181, 193
GlobalMax 35
GlobalPos 35
GlobalRHSBinds 181
globalRtnUnknown 104, 106
GlobalSalience 111, 114
GlobalString 35
gm1 12
gm2 12
gm3 12
GrabGenericWildargs 295
GrabWildargs 286
grad-deg 273
GroupActions 55
GroupHandlerWildargs 373
GroupPatterns 114
GSMP_LEN 380
GSM_PREFIX 380
halt 205
Index

HaltExecution 58, 59, 60
HaltRules 161
HandlerDeleteError 386
HandlerDeleteSlot 360, 373
HandlerGetBind 370, 373, 387
HandlerGetSlot 374
HandlerInsertSlot 360, 374
HandlerPutSlot 362, 374
HandlerReplaceSlot 374
HandlerRtnUnknown 370, 374
HandlersExecuting 386
HandlerType 387
HandlerWildargs 370, 375
HANDLER_DECL 319
HashClass 333, 341
HashFact 90
HashFloat 22
HashInstance 357, 366
HashInteger 22
HashSymbol 23
HashTablestoCode 257
HasSuperclass 322
HeaderFP 253
help 275
help-path 275
HelpFunctionDefinitions 275
HIGHER_PRECEDENCE 301
hndquals 380
IDENTICAL 301
IdenticalExpression 45
if 261
IfParse 51
IgnoreCompletionErrors 35
ImageID 253
IncrementalReset 157, 158, 200
IncrementalResetFlag 157
IncrementFactCount 90
IncrementFloatCount 23
IncrementIndentDepth 38
IncrementInstanceCount 362
IncrementIntegerCount 23
IncrementPseudoFactIndex 183
IncrementSymbolCount 23
IndentationDepth 36, 38
INDIRECT 334
INHERIT_BIT 319
init-slots 362, 377
InitImage 184, 254, 259
InitFactCommands 93
InitGenModule 142
initialize-instance 345, 346, 351, 354, 362, 363
InitializeAtomTables 23
InitializeBlockMemory 16
InitializeClasses 336
InitializeCLIPS 6
InitializeConstructs 71
InitializeDefaultRouters 31
InitializeDeffacts 97
InitializeDeffunctions 284
InitializeDefglobal 104
InitializeDefinstances 415
InitializeDefrules 198
InitializeDeftemplateHashTable 214
InitializeDeftemplates 211
InitializeEngine 165, 198
InitializeFactHashTable 92
InitializeFacts 90
InitializeFunctionHashTable 48
InitializeIgnoredConstructs 71
InitializeInstance 351, 361, 363
InitializeInstanceTable 364
InitializeNonportableFeatures 6
InitializeSpecialForms 49
INIT_STRING 370
INIT_SYMBOL 370
inscom.c 4, 345
INSERT 358, 360
InsertHandlerHeader 376
InsertSlot 336
InsertSlotOverrrides 366
insfun.c 4, 357
insfun.h 358
insquery.c 4, 395
insquery.h 401, 402
InstallClass 341, 377
InstallDeftemplate 208, 214
InstallExpression 47, 214
InstallFunctionList 46
InstallInstance 367
instance 345, 357
instance-address 348
instance-addressp 349
instance-existp 348
instance-name 349
instance-name-to-symbol 349
instance-namep 349
instance-set 395
distributed action 395
query 395
Instance Template Codes 334
InstanceGarbageList 359
InstanceList 346, 357
InstanceListBottom 359
InstanceLocationInfo 367
InstanceNameToSymbol 349
instancep 349
instances 316, 347
InstanceSizeHeuristic 367
InstancesPurge 337
INSTANCE Table 346, 360
INSTANCE_SLOT_REF 398
INSTANCE_TABLE_HASH_SIZE 357
integer 267
IntegerArray 244
integerp 263
IntegersToCode 258
IntegerTable 18, 19, 20, 21, 22, 23, 24, 25, 26, 39, 240, 241, 249, 256, 257, 258, 259
intrfile.c 277
IOFunctionDefinitions 265
IsClassAbstract 322
IsClassAbstractCmd 322
IsClassBeingUsed 337
IsDefclassDeletable 322
IsDeffactsDeletable 97
IsDeffunctionDeletable 284
IsDefgenericDeletable 292
IsDefinstancesDeletable 414
IsDefmessageHandlerDeletable 372
IsDefmethodDeletable 292
IsDefruleDeletable 198
IsDeftemplateDeletable 211
IsInstance 349
IsInstanceAddress 349
IsInstanceName 349
IsMethodApplicable 306, 308
IsQueryFunction 403
IsSlotBound 350, 353
IsSlotInitable 350, 353
IsSlotWritable 350, 353
IsSubclass 322, 324
IsSuperclass 323, 324
IsSystemClassName 337
JoinArray 203
JoinHandlerLinks 390, 391, 392
JoinNetErrorMessage 159
LastDeffacts 95
LastDefrule 195, 199
LastDeftemplate 207
LastFact 86
length 267
lexemep 263
lgcldpnd.c 2, 187
LHSError 112
LHSpattern 114
list-defclasses 321
list-deffacts 97
list-deffunctions 282
list-defgenerics 291
list-definstances 414
list-defmessage-handlers 372
list-defmethods 291
list-deftemplates 207, 212
list-defglobals 105
list-deftemplates 213
ListAgenda 165
ListBreakpoints 165
ListDefclasses 322
ListDeffacts 97
ListDeffactsCommand 97
MergePartialMatches 183
MergeSlots 329, 334, 341
message 369, 379
delete 352, 360, 370, 376, 415
init 361, 362, 370, 376, 384
print 376
put 361, 366
message-handler 261, 369, 379
types
after 380, 388, 390
around 380, 382, 390, 392
before 380, 388, 390
primary 380, 382, 388, 389, 390, 391
Message-Handler Lookup Codes 380
message-handler precedence 381
Message-Handler Type Codes 380
message dispatch 337, 379, 384
message parameter array 370, 373, 374, 375, 378, 379, 389, 392
Message Trace Strings 381
method 261, 290, 299
method parameter array 290, 297, 300, 304, 307
method precedence 299, 301, 307
Method Precedence Codes 301
MethodAlterError 305
MethodsExecuting 305
min 51, 267
mod 273
ModAndDupParse 218, 219, 221
modify 3, 54, 201, 207, 212, 213, 214, 217, 219, 220, 221
ModifyCommand 212
ModifyParse 219
ModifySlotsMultiplyDefined 221
MoveActivationToTop 165
msgcom.c 4, 369
msgcom.h 370
msgfun.c 4, 379
msgfun.h 380
MultiArgNumericParse 51
MultiArgParse 51
Multifield Slot Function Codes 358
MultifieldFunctionDefinitions 269
multieldp 263
MultifieldSlotDelete 350, 353, 360
MultifieldSlotInsert 350, 353, 360
MultifieldSlotReplace 351, 353, 360
MultiIntoSingleFieldSlotError 219
multiple inheritance 329, 339, 401
MultiplyDefinedLHSSlots 230, 231
MultiplyDefinedSlots 226
multivar.c 3, 269
MULTI_CLEAR 358, 362
MULTI_SET 358, 362
mv-append 269
mv-delete 269
mv-replace 269
mv-slot-delete 350
mv-slot-insert 351
mv-slot-replace 351
mv-append 2
mv-subseq 269
NegEntryRetract 180
neq 135, 136, 263
neqfield 135, 136
neqvar 135
neq_field 137, 261
neq_vars 158
NetworkPointer 145
NetworkRetract 179
NewCFile 255
NewClass 337
NewGeneric 308
NewHandler 387
NewInstance 367
NewPseudoFactPartialMatch 183
NewSlot 337
NewSystemHandler 376
next-handlerp 387
next-methodp 305
NextFactID 92
NextFactIndex 86
NextHandlerAvailable 387
NextInCore 301, 302, 381, 382, 388
NextMethodP 305
NoInstanceError 364
nonconstant 261
NonexistantError 60
nop 137, 261
not 52, 136, 137, 263
not CE 109, 114, 115, 117, 127, 149, 177
notconstant 135, 137
NotParse 52
NotPatternParse 115
nth 269
NumberOfActivations 162, 164
NumberOfDeffacts 95
NumberOfDefglobals 101, 104
NumberOfDefrules 203
NumberOfDeftemplates 208
NumberOfExpressions 245, 248
NumberOfFacts 86, 88, 89
NumberOfJoins 203
NumberOfPatternPointers 203
NumberOfPatterns 203
NumberOfTemplateSlots 208
numberp 263
objc.c 4, 419
objcmp.c 4, 417
object.h 397, 399
ObjectParseToken 319, 347
ObjectsRunTimeInitialize 323
ObjectSystemPurge 338, 340
OBJECT_CLASS_STRING 333
oddp 263
OldGenericBusySave 301, 308, 309
open 75, 265
OpenErrorMessage 82
OpenFile 29, 31
OpenStringDestination 29, 31
OpenStringSource 29, 31
OpenTextSource 32
options 267
or 51, 137, 158, 263
or CE 109, 113, 114, 117
OutOfMemoryFunction 9
override-next-handler 381, 387
OverrideSlotProtection 358
PackExpression 46, 47
PackRestrictionTypes 295
PackSlots 338
ParseAllowedValuesAttribute 226
ParseAssertSlotValues 221
ParseAssertTemplate 55, 219, 222
ParseAtomOrExpression 46, 108
ParseConstantArguments 46
ParseConstruct 72
ParsedBindNamesEmpty 50
ParseDefault 226
ParseDefclass 317, 325, 326, 327, 334, 335, 336, 337, 338
ParseDefclassName 326
ParseDeffacts 98
ParseDeffunction 284, 286, 287
ParseDefgeneric 295, 296, 301
ParseDefglobal 108
ParseDefinstances 415
ParseDefinstancesName 415
ParseDefmessageHandler 341, 371, 376, 377, 378, 385
ParseDefmethod 296, 297, 301, 302, 303, 304
ParseDefrule 200
ParseDeftemplate 225
ParseHandlerParameters 377
ParseIgnoredConstruct 72
ParseInitializeInstance 347, 351, 354, 363, 364
ParseMethodName 296
ParseMethodNameAndIndex 296
ParseParameters 286, 294, 296
ParseQueryAction 396, 404, 405, 406, 407
ParseQueryActionExpression 405
ParseQueryNoAction 396, 405, 406, 407
ParseQueryRestrictions 405, 406, 407, 408
ParseQueryTestExpression 406
ParseRangeAttribute 226, 227
ParseRestriction 295, 296, 298
<table>
<thead>
<tr>
<th>ParseRuleLHS 112, 200</th>
<th>PPBackupTwice 36</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParseRuleRHS 112, 200</td>
<td>PPBackupMax 36</td>
</tr>
<tr>
<td>ParseSimpleInstance 347, 353</td>
<td>PPBackupPos 36</td>
</tr>
<tr>
<td>ParseSlot 227, 228, 326</td>
<td>PPBackupStatus 36, 38</td>
</tr>
<tr>
<td>ParseSlotLabel 222</td>
<td>PPCRAndIndent 38</td>
</tr>
<tr>
<td>ParseSlotOverrides 354</td>
<td>PPDefclass 323, 324, 326</td>
</tr>
<tr>
<td>ParseSlotValue 326</td>
<td>pdeffacts 97</td>
</tr>
<tr>
<td>ParseSuperclasses 327</td>
<td>pdeffactsCommand 97</td>
</tr>
<tr>
<td>ParseException 347</td>
<td>ppdeffunction 282</td>
</tr>
<tr>
<td>ParseTypeAttribute 226, 227</td>
<td>PPDefgeneric 292, 293</td>
</tr>
<tr>
<td>ParsingTopLevelCommand 64, 67</td>
<td>ppdefglobal 105</td>
</tr>
<tr>
<td>parsutil.c 2, 53</td>
<td>PPDefglobalCommand 105</td>
</tr>
<tr>
<td>PatPtrArray 203</td>
<td>PPDefinstances 414</td>
</tr>
<tr>
<td>pattern CE 109, 113, 114, 149, 177</td>
<td>pdefmessage-handler 375</td>
</tr>
<tr>
<td>PatternArray 203, 204</td>
<td>PPDefmessageHandler 375</td>
</tr>
<tr>
<td>PatternHasTemplate 125</td>
<td>PPDefmethod 293</td>
</tr>
<tr>
<td>PatternMatch 174</td>
<td>ppdefrule 79, 205</td>
</tr>
<tr>
<td>PatternNetErrorMessage 174</td>
<td>pdeftemplate 207, 212, 213</td>
</tr>
<tr>
<td>PatternNetworkPointer 144, 145</td>
<td>PPDeftemplateCommand 212</td>
</tr>
<tr>
<td>PCALL 279</td>
<td>PPDrive 157, 159</td>
</tr>
<tr>
<td>PERFORM 381</td>
<td>PPInstance 352, 354, 377</td>
</tr>
<tr>
<td>PerformImplicitHandler 382, 391</td>
<td>PredicateFunctionDefinitions 263</td>
</tr>
<tr>
<td>PerformMessage 383, 384, 388, 389, 391</td>
<td>PreserveEscapedCharacters 77</td>
</tr>
<tr>
<td>PeriodicCleanup 77, 82, 361</td>
<td>PrettyPrintBuffer 35, 36, 37, 38</td>
</tr>
<tr>
<td>pi 273</td>
<td>PREVIEW 381</td>
</tr>
<tr>
<td>PlaceActivation 163, 168</td>
<td>preview-generic 306</td>
</tr>
<tr>
<td>PlaceBreadthActivation 168</td>
<td>preview-send 375</td>
</tr>
<tr>
<td>PlaceComplexityActivation 168</td>
<td>PreviewGeneric 305, 307</td>
</tr>
<tr>
<td>PlaceDepthActivation 169</td>
<td>PreviewMessage 372, 383, 385, 386</td>
</tr>
<tr>
<td>PlaceLEXActivation 169</td>
<td>PreviewMessageCmd 375</td>
</tr>
<tr>
<td>PlaceMEActivation 169</td>
<td>PrimeJoin 184, 200</td>
</tr>
<tr>
<td>PlacePattern 146</td>
<td>PrimitiveClassMap 332, 333</td>
</tr>
<tr>
<td>PlaceRandomActivation 169</td>
<td>print-region 275</td>
</tr>
<tr>
<td>PlaceSimplicityActivation 169</td>
<td>PrintAbbreviatedHandlerRemoval 387</td>
</tr>
<tr>
<td>PNLDrive 157, 158</td>
<td>PrintActivation 165</td>
</tr>
<tr>
<td>PNRDrive 158, 159</td>
<td>PrintAtom 82</td>
</tr>
<tr>
<td>pointer 261</td>
<td>PrintClassBrowse 327</td>
</tr>
<tr>
<td>interp 263</td>
<td>PrintCLIPS 32</td>
</tr>
<tr>
<td>PoolSize 13</td>
<td>PrintCRSVA ctivation 166</td>
</tr>
<tr>
<td>PopQueryCore 407</td>
<td>PrintCString 258</td>
</tr>
<tr>
<td>PosEntryRetract 179</td>
<td>PrintCurrentMessage 387, 388</td>
</tr>
<tr>
<td>PPBackup 36, 38</td>
<td>PrintDataObject 59</td>
</tr>
<tr>
<td>PPBackupOnce 36</td>
<td>PrintDeffunctionReference 255</td>
</tr>
</tbody>
</table>
PrintExpression 47
PrintFact 90
PrintFactWithIdentifier 90
PrintFloat 82
PrintFloatReference 252, 255
PrintFunctionReference 255
PrintGenericFunctionReference 256
PrintHandler 388
PrintInChunks 82
PrintInstance 354
PrintIntegerReference 252, 256
PrintLongInteger 83
PrintMethod 306
PrintNoHandlerError 392
printout 265
PrintPartialMatch 184
PrintPartialOrderLoop 331, 342
PrintPreviewHandler 388, 392
PrintPrompt 66
PrintSymbolReference 252, 256
PrintTally 83
PrintTemplateFact 212
PrintWhileLoading 70, 71, 74
PRINT_STRING 371
progn 53, 55, 233, 261, 279, 299, 369
PropagateReturnValue 59
PropogateReturnValueValue 282, 305, 361, 392
PseudoFactIndex 181, 183, 185
PSMP_LEN 380
PSM_PREFIX 380
PTR_AND 136
PTR_CONSTANT 136
PTR_EQ 136
PTR_EQ_FIELD 136
PTR_GET_FIELD 136
PTR_NEQ 136
PTR_NEQ_FIELD 137
PTR_NOP 137
PTR_NOT 137
PTR_NOTCONSTANT 137
PTR_OR 137
PurgeUserClassStuff 327
PushQueryCore 407
put 374
PutClassInTable 338
PutSlotValue 358, 364
QFindDefglobal 105
QFindDeftemplate 212
QGetDefglobalValue 105
QSetDefglobalValue 105
QSetListOfDeftemplates 212
QuashInstance 365, 367
QueryCore 398, 403, 407
QueryCoreStack 398, 403, 407
QueryDoForAllInstances 400, 408
QueryDoForInstance 400, 410
QueryFindAllInstances 400, 402, 408
QueryFindInstance 401, 410
QueryRouter 33
QueryRouters 32
QUERY_DELIMITER_STRING 397
QUERY_DELIMITER_SYMBOL 397, 402
rad-deg 273
random 267
read 265
readline 265
ReadNeededFloats 249
ReadNeededFunctions 249
ReadNeededIntegers 249
ReadNeededSymbols 250
ReadUntilClosingParen 55
RecordPartialOrder 342
refresh 205
refresh-agenda 205
RefreshAgenda 166
RefreshBooleanSymbols 24
RefreshDefrule 198
RefreshExpressions 250
ReinitializeClasses 338
release-mem 267
ReleaseMemory 13
ReleaseTraversalID 316, 397, 401
RememberJoinsForRule 201
remove-break 205
RemoveActivation 166
RemoveAllActivations 166
RemoveAllBreakpoints 166
RemoveAllDeffacts 97
RemoveAllFacts 90
RemoveBreakpoint 166
RemoveCleanupFunction 83, 84
RemoveClearFunction 72
RemoveConstruct 72
RemoveCPIFunction 84
RemoveDeffunction 287
RemoveDefinstances 415
RemoveEphemeralAtoms 24
RemoveEphemeralFloats 24, 25
RemoveEphemeralIntegers 24, 25
RemoveEphemeralSymbols 24, 26
RemoveFactDependencies 192
RemoveFloat 26
RemoveFunctionParser 47
RemoveGeneric 297
RemoveGenericMethod 297
RemoveHashDeftemplate 215
RemoveHashedFact 90
RemoveInteger 26
RemoveIntranetworkLink 146
RemoveOldFacts 91
RemovePeriodicFunction 83, 84
RemovePMDependencies 192
RemoveResetFunction 73
RemoveRuleNetwork 200
RemoveRunFunction 166
RemoveSaveFunction 73
RemoveSymbol 26
reorder.c 2, 117
ReorderAgenda 166, 167
ReorderAssertSlotValues 222
ReorderLHSSlotValues 230, 231
ReorderPatterns 118
REPLACE 358, 360
ReplaceExpressionVariables 201
ReplaceGlobalVariable 106, 201
ReplaceHandlerParameters 376, 378
ReplaceInstanceVariables 399, 403, 407, 408
ReplaceParameters 287, 295, 297
ReplaceRHSVariable 201
ReplaceSlotReference 399, 408
RequestChunk 10, 13, 16
RerouteStdin 6
reset 92, 97, 102, 104, 198, 261, 415
ResetCLIPS 73
ResetCompositeSlots 342
ResetDeffacts 98
ResetDefglobals 106
ResetDefrules 201
ResetDeployedRuleImage 184, 185
ResetFacts 92
ResetGlobals 102, 104, 106
ResetNotedJoin 185
ResetNotedPatterns 185
RestrictAllWatchItems 83
RestrictBusyCount 308
RestrictionParse 112, 231
RestrictionsCompare 308, 309
reteutil.c 2, 181
retract 93
retract.c 2, 177
RetractFact 91, 191
RetractParse 52
return 261, 279, 282, 299, 304, 369, 391
ReturnChunk 10, 13
ReturnContext 261
ReturnDefrule 198
ReturnElements 91
ReturnExpression 47
ReturnFlag 261, 282, 302, 305, 382, 388, 392
ReturnMarkers 180
ReturnNodes 119
ReturnPackedExpression 47
ReturnPartialMatch 179
ReturnSAPs 223
ReturnSlots 213
ReturnSLPs 231
ReturnSymbolMatches 24
ReturnValues 59
ReuseJoin 146
ReverseOR 119
Index

rm 14
rm3 14
round 273
RouteCommand 66
router.c 1, 27
RtnGenericUnknown 290, 293
rtn_struct 14
rtn_var_struct 14
RuleAnalysis 131
RuleBodyParse 115
rulecom.c 3, 205
RuleComplexity 125
RuleDeletions 197, 199
RuleFiring 162, 165
ruleprsr.c 2, 109
rules 205
run 162, 205
RunCLIPS 167
SalienceEvaluation 162, 165, 167
SalienceExpression 111, 114
SalienceInformationError 199
save 73, 97, 104, 198, 207, 211, 277, 287, 297, 327, 416
save-instances 352
save-facts 76, 93
SaveAllWatchItems 83
SaveBusyCount 309
SaveConstructs 73
SaveDefclasses 327
SaveDeffacts 98
SaveDeffunctionHeaders 287
SaveDeffunctions 287
$SaveDefgenerics 297
SaveDefglobals 108
SaveDefinstances 416
SaveDefmethods 297
SaveDefrules 202
SaveDeftemplates 215
SaveInstances 348
SaveInstancesCommand 352
SavePPBuffer 38
scanner.c 1, 35
ScanNumber 39
ScanString 39
ScanSymbol 39
SDDCommand 213
SearchParsedBindNames 50
sec 273
sech 273
SecondaryFunctionDefinitions 267
seed 267
SegmentDeinstall 88, 91
SegmentInstall 88, 91
SELF_LEN 371
SELF SLOT REF 371
SELF STRING 371
SELF_SYMBOL 371
send 375, 379, 384
SequenceRestrictionParse 115
set-dynamic-deftemplate-checking 214
set-salience-evaluation 205
set-auto-float-dividend 263
set-break 205
set-dynamic-deftemplate-checking 207, 213
set-fact-duplication 93
set-incremental-reset 205
set-reset-globals 104, 107
set-strategy 205
SetActivationSalience 167
SetAgendaChanged 167
SetAllWatchItems 83
SetBeforeClearFunction 73
SetBeforeResetFunction 73
setbit 398, 401
SetClassList 338
SetCommandString 66
SetCompilationsWatch 73
SetConserveMemory 14
SetDefglobalValue 105, 106
SetDefinstancesList 414
SetDynamicDeftemplateChecking 213
SetEvaluationError 60
SetEventFunction 66
SetExecutingConstruct 74
SetFactDuplication 91
SetFactID 92
SetFactListChanged 92
SetFastLoad 32
SetFastSave 32
SetFloatTable 24
SetFunctionList 47
setgen 267
SetGenericList 306
SetGlobalsChanged 106
SetHaltExecution 60
SetIncrementalReset 158
SetIndentDepth 38
SetInstancesChanged 357, 362
SetIntegerTable 24
SetListOfDeffacts 97
SetListOfDeffunctions 284
SetListOfDefglobals 106
SetListOfDefrules 199
SetListOfDeftemplates 213
SetMemoryStatusFunction 66
SetMultifieldErrorValue 60
SetNetworkPointer 145
SetOutOfMemoryFunction 15
SetParsedBindNames 50
SetPPBufferStatus 38
SetPrintWhileLoading 74
SetPseudoFactIndex 185
SetResetGlobals 106
SetResetGlobalsCommand 107
SetRuleDeletions 199
SetRuleInformation 125, 132
SetSalienceEvaluation 167
SetStrategy 167
SetSymbolTable 24
SetTraversalID 316, 397, 401
SetupClasses 323
SetupDefinstances 414
SetupGenericFunctions 293
SetUpInitFile 258
SetupInstances 352, 364
SetupMessageHandlers 371, 375
SetupObjectSystem 323, 352, 401, 414
SetupQuery 401
SetVariableInformation 125, 132
SetWatchItem 83
show-breaks 205
signal 5
SimplePatternParse 115
sin 273
sinh 273
SizeOfDefglobalArray 102
SizeOfglobalArray 107
slot-accessor 18, 371, 380, 382, 390, 391
Slot-Accessor Prefix Strings 380
slot-boundp 350
slot-existp 348
slot-facets 324
slot-initablep 350
slot-override 351, 354, 361, 362, 366
slot-sources 324
slot-writablep 350
Slot Value Expression Evaluation Codes 358
Slot Value Set Codes 358
SlotArray 208
SlotDeclarations 225, 227
SlotExistError 365
SlotFacetsCmd 323, 324
SlotSourcesCmd 324
SlotValueExpression 365
SLOT_COMPOSITE_RLN 320
SLOT_DEF_DYN_RLN 319
SLOT_DEF_RLN 319
SLOT_EMPTY 358, 364
SLOT_ERROR 358, 364
SLOT_EXCLUSIVE_RLN 320
SLOT_FILLED 358, 364
SLOT_INH_RLN 319
SLOT_INIT_RLN 320
SLOT_LOCAL_RLN 320
SLOT_MULT_RLN 320
SLOT_NOINH_RLN 319
SLOT_RDONLY_RLN 319
SLOT_RDONLY_RLN 319
SLOT_RDWRT_RLN 319
SLOT_RLN 319
SLOT_SGL_RLN 320
SLOT_SHARE_RLN 320
SortBindings 170
spclform.c 1, 41, 49
sprintf 80
sqrt 273
StaleInstanceAddress 365
STORAGE_BIT 319
StoreSubclasses 327
StoreValuesInMultifield 368
str-assert 267
str-cat 271
str-compare 271
str-explode 269
str-implode 269
str-index 271
str-length 271
Strategy 162, 165, 167
StringBuffer 40
StringFunctionDefinitions 271
stringp 263
StringPrintForm 38
strings.c 3, 271
StringToFact 87, 92
StringToMultifield 92
sub-string 271
subclassp 323
subset 269
superclassp 323
SUPERCLASS_RLN 319
sym-cat 271
symbol-to-instance-name 352
symbol.c 1, 17, 75
SymbolArray 244
symbolp 263
SymbolsToCode 259
SymbolTable 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 35, 39, 40, 214, 240, 241, 250, 256, 257, 258, 259
SymbolToInstanceName 352
SyntaxErrorMessage 84
sysdep.c 1, 5
sysio.c 3, 265
syspred.c 3, 263
sysprime.c 3, 261
syssecnd.c 3, 267
system 267
SystemFunctionDefinitions 6
SYSTEM_NO 300
SYSTEM_YES 300
TabulateInstances 354
TagLHSLogicalNodes 115
TagRuleNetwork 185
tan 273
tanh 273
TemplateMultifieldSlotReplace 215
TempMemoryPtr 8, 11, 12, 14
TempSize 8, 14
test CE 109, 114, 116, 130, 177
testbit 398, 402
TestEntireChain 396, 408, 409
TestEntireClass 396, 402, 408, 409
TestForFirstInChain 396, 410, 411
TestForFirstInstanceInClass 396, 410, 411
TestPattern 116
TestTraversalID 316, 397, 401
textpro.c 3, 275
TheLogicalJoin 161, 187, 191, 193
time 267
TopLevelCommand 66
TopMemoryBlock 9
TopNegJoinRetract 180
TopOfCore 301, 307, 381, 391
toss 275
TraceDeffunction 287
TraceErrorToPattern 174, 175
TraceErrorToRule 174, 175
TraceGeneric 309
TraceHandler 388, 392
TraceMessage 393
TraceMethod 309
TRAVERSAL_BYTES 315, 337, 397
TrueSymbol 18, 20
trunc 267
type 293, 348
TypeListCompare 309
TypeOf 293
undefclass 321
undeffacts 97, 98
UndeffactsCommand 98
undeffunction 282
undefgeneric 291
undefinstances 414
undefmessage-handler 372
undefmethod 291
undefrule 79, 205
undeftemplate 207, 213
UndeftemplateCommand 213
UngetCLIPS 33
unmake-instance 352
UnmakelInstance 352
UnmarkBuckets 241, 259
UnrecognizedRouterMessage 33
unwatch 261
upcase 271
UpdateMemoryRequests 15
UpdateMemoryUsed 15
UpdateModifyDuplicate 201, 218, 219, 220
UserFunctions 421
USER_CLASS_STRING 333
utility.c 2, 75
ValidConstruct 74
ValidInstanceAddress 348
ValidSlotValue 366
ValidType 298
ValueDeinstall 60
ValuelInstall 60
variable.c 2, 121
VariableAnalysis 131
VersionString 65
watch 2, 75, 77, 261, 281, 300, 359, 364, 380
WatchActivations 162
WatchCompilations 70, 71, 73
WatchDeffunctions 281
WatchFacts 86
WatchGenerics 300
WatchGlobals 102
WatchHandlers 380
WatchInstances 359
WatchMessages 380
WatchMethods 300
WatchRules 195, 197
WatchSlots 359
WatchStatistics 162
WCLIPS 27
WDIALOG 27
WDISPLAY 27, 90
WERROR 27
while 261
WhileParse 52
wildcard parameter 279, 290, 369
WildDeleteHandler 372, 383
WithinInit 359
WriteBinaryFooter 241
WriteBinaryHeader 241
WriteNeededFloats 241
WriteNeededFunctions 242
WriteNeededIntegers 241
WriteNeededSymbols 242
WrongTypeError 61
WTRACE 27