expeval: Expression Evaluation API Reference

CML00052-01

Code Magus Limited (England reg. no. 4024745)
Number 6, 69 Woodstock Road
Oxford, OX2 6EY, United Kingdom
www.codemagus.com
Copyright © 2014 – 2024 by Code Magus Limited
All rights reserved

July 30, 2024

1 Introduction

The `expeval` library provides a means of representing an expression and a means of evaluating that expression. The expression library also provides a means by which variables can be supplied by the user of the library. The library also provides means by which the caller the user of the library can define their own functions which can appear in an expression. All variables used in the expressions need to be supplied by the user of the library. Additionally, any number of sources of variables can appear in a single expression. Expressions can also refer to string and numeric literal values for which the library supplies transfer mechanisms to represent the literals in a format upon which the operators of the expressions can execute.

For our purposes an expression is a acyclic tree rooted in a node which represents the entire expression. The nodes of the expression-tree can be classified as interior or terminal nodes. There is always a single root node which may degenerately be a terminal node or may in turn refer to other nodes and hence be classified as an internal node. Terminal nodes represent variables and literals, whilst internal nodes represent operators and function calls.

There can be any number of child nodes of any node in the expression tree ranging from no children (leaf nodes) to any arbitrary number of nodes. For non-terminal nodes, the number of nodes needs to make sense to the type of the parent node. For monadic arithmetic and logical operators, there should only be one child node. For dyadic arithmetic and logical operators the number of children should be two. For functions of n arguments, there should be n children.

The data structure that achieves the arbitrary edge-degree is the child-peer structure where the first child is pointed to from the parent node as the `child` of the node. The second child or first sibling to the first child is pointed to from the first child node as the `peer` of the node. The second sibling or third child is then pointed to by the `peer` of the second child. The chain of children is then terminated will a `NULL` `peer` pointer, and the situation of no children is indicated by a `NULL` value of the parent's `child` pointer.

The intention for this tree structure is that it naturally fits an abstract syntax notation tree representation of a parsed expression. Figure 1 shows such a tree representing the expression:

$$1.3 + f(a, b, 4) * 10$$

In this expression-tree, the nodes have the following meaning:

- + and * These nodes are internal nodes representing arithmetic operators. Internal nodes always have children nodes and these children nodes represent the operands of the operator.
- f This node is also an interior node and represents the application of the function f on its operands (if any). Function nodes may or may not have child nodes, and whilst this is a special case, it does not change the general processing of the node.

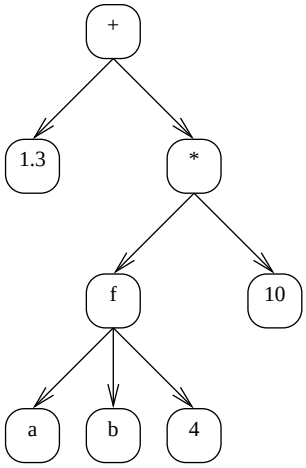


Figure 1: Abstract Syntax Tree Representation of the Expression $1.3 + f(a, b, 4) * 10$

In general, the children nodes of function internal nodes represent the evaluations of the argument expressions to be passed to the function. The `expeval` library does not have predefined functions and the functions need to be introduced to the library so that their evaluation can be executed during the evaluation of any expressions that contain them.

- a and b These nodes are terminal nodes that represent variables. The `expeval` library does not define any variables, but rather provides a means by which suppliers of variable values can be introduced to the library.
- 1.3, 4 and 10 These nodes are terminal nodes that represent literals. In this case the literals are all numeric, but string may also be represented. The `expeval` library provides the means of introducing the literals values to the library.

In addition to the above and in support of the interfacing required by the creator of expression trees, the `expeval` library provides a number of utility and transfer functions. These functions are used to create nodes, insert values, extract values, define variable suppliers, define functions and to evaluate the expressions.

The remainder of this document describes the usage of the `expeval` library with examples.

2 `expeval` Contexts

All the required data structures and function definitions are made available to the user of the `expeval` library by including the header:

```
#include <expeval.h>
```

Expressions are created and evaluated within a context. In general before any processing using the expression evaluation library, a context must be created. The context is a structure returned to the caller of the library and contains certain information such as default encodings, a means of reporting errors to the caller, a means of providing an anchor for defined functions and variable suppliers. The context is passed back to all further calls to the expression evaluation library.

The following fragment shows how a context is declared and created.

```
expeval_context_t *context;   /* expeval library context */

context = expeval_context_open(EVALFL_ASCII);
```

In the above example, the context is created and the returned pointer is placed in the variable `context`. The only argument function `expeval_context_open` is a set of flags, in this case only the `EVALFL_ASCII` is passed, indicating that character data is encoded using an ASCII/ISO-8 based code page. Once a context has been created, the option flags can be adjusted using the function `expeval_context_flags`. See the header file `expeval.h` for the definition of additional flags that can be passed to the functions `expeval_context_open` and `expeval_context_flags`.

The prototypes of the functions required to create a context, to free a context and to change the flag options used by a context are:

```
expeval_context_t *expeval_context_open(unsigned long flags);
void expeval_context_close(expeval_context_t *context);
int expeval_context_flags(expeval_context_t *context, unsigned long flags);
```

Once the `expeval` context is no longer required, it can be freed using the `expeval_context_close` function. This function drives the cleanup of any resources that were acquired during the life of the context and that are associated as being owned by the context. This includes any defined functions, including any resources that have been accumulated against the definition of a particular function; and also includes any variable suppliers and the resources acquired by them.

```
expeval_context_close(context);
```

3 Building Literal Leaf Nodes

In general there are three types of value that are supported by the `expeval` library:

- Numbers The value space of numbers includes all positive and negative numbers which can be represented with a precision of a certain number of decimal digits. This number is defined in the header file `expeval.h`:

  ```
  #define EXPEVAL_DECIMAL_DIGITS 32
  ```

- Strings In general a string in the `expeval` library refers to a piece of storage with a fixed length. However, the transfer functions for inserting and extracting strings from the library allow the strings to be externally defined by null-terminated strings or pieces of storage with a supplied length. The encoding of string data as represented as storage is determined the flags supplied on the creation of the context being used and if no indication of the encoding (`EVALFL_ASCII` or `EVALFL_EBCDIC`) is supplied, then the hosting machine is inspected and the base encoding of that machine is used.
- Booleans These values represent *true* or *false*.

Literals can be created which represent the values of the above types. Literal leaf nodes are created as two-step process. The first step creates a value of the required type.

Assuming the following definitions:

```
int rc;
expeval_value_t value;
expeval_node_t *node;
```

The value creation is accomplished by one of the following function calls, depending on the type of value being created:

```
   rc = expeval_value_number(context,"123.45",&value);
   rc = expeval_value_string(context,"Coffee Shop",&value);
   rc = expeval_value_boolean(context,"0",&value);
```

These functions and their variants, have the following prototypes:

```
int expeval_value_string(expeval_context_t *context, unsigned char *string,
      expeval_value_t *value);
int expeval_value_homest(expeval_context_t *context, unsigned char *string,
      expeval_value_t *value);
int expeval_value_hexdata(expeval_context_t *context, unsigned char *string,
      expeval_value_t *value);
int expeval_value_number(expeval_context_t *context, unsigned char *number,
      expeval_value_t *value);
int expeval_value_boolean(expeval_context_t *context, unsigned char *boolean,
      expeval_value_t *value);
```

If one of these functions fails to insert the literal value in to the supplied literal structure, then the function, like many of the `int` returning functions where a return value of `-1` indicates a failure and a value of `0` indicates success. On failure a `expeval` library function details the failure by placing a formatted message into the `last_error` member of the context structure used:

```
   rc = expeval_value_boolean(context,boolean,&value);
   if (rc)
      {
      printf("In check_build_boolean_lit:\n");
      printf("   expeval_value_boolean = %d for %s .\n",rc,boolean);
```

```
    printf("   Error = %s\n",context->last_error);
    return NULL;
    }
```

The node with the required value can be created using the `expeval_build_lit_leaf` function which has the prototype:

```
expeval_node_t *expeval_build_lit_leaf(expeval_context_t *context,
      expeval_value_t *value);
```

This value can then be used to create a terminal node for later inclusion into an expression tree, using the following fragment:

```
node = expeval_build_lit_leaf(context,&value);
```

Cleanup of a literal nodes is accomplished by performing a cleanup of the expression from a root node downwards. In the above example, the one node expression rooted in `node` can release all the resources acquired by making the following call:

```
expeval_cleanup(node);
```

4 Building Variable Leaf Nodes

The process of building variable leaf nodes is little more involved than building literal leaf nodes. For variables to be resolvable, there needs to be a binding between the variable name and the method that is going to supply the type of the variable (during expression type checking) and the value of the variable (during expression evaluation). The binding requires the notion of a variable supplier.

Variables have names which resemble a dotted qualification scheme which includes possible array indexing. The meaning of any indexing and the exact method of expressing the indexing is up to the variable supplier to interpret and the parser that is being used. The only dependency for the `expeval` library is that the variable name has a node structure where the first node (if it is not the only node) is delimited by one of the characters '`.`' (point), '`[`' (open bracket) or '`(`' (open parenthesis). This is only a requirement if a specific variable supplier is to be used for the variable binding. It is also possible to assign a generic variable supplier for the binding by placing the structure describing the binding in the context being used (see `default_var_supplier` in the context structure in `expeval.h`. See below for details on creating variable suppliers.

In the following fragment a variable supplier is introduced to supply the bindings for types and variables that consist entirely of, or have as there first node, the identifier `TEST-MAP`. In this example, the supplier is also being made the default supplier:

```
expeval_supplier_t *supplier;       /* supplier for default var binding */

supplier = expeval_varsup_open(context,"TEST-MAP",symvar_resolve,
   symvar_byteimage,
```

```
      symvar_value,symbols);

   if (!supplier)
      {
      fprintf(stderr,
         "Failure to create supplier: %s.\n",
         context->last_error);
      return -1;
      }

   context->default_var_supplier = supplier;
```

Once a variable supplier has been created to bind variables, variable leaf nodes can be created. The following creates a variable leaf node using the variable supplier defined above:

```
   expeval_node_t *var;

   var = expeval_build_var_leaf(context,strdup("TEST-MAP.TEST-FIELD-3"));
   if (!var)
      fprintf(stderr,
            "Error in expeval_build_var_leaf: %s\n",
            context->last_error);
```

As mentioned above, the `expeval` library use the high-level node of the variable name to determine which variable supplier is given control to perform the binding. Thus the call to `expeval_build_var_leaf` results in the `symvar_resolve` function being called to perform the variable resolution and the binding.

```
  /*
   * Local prototypes:
   */

static void cleanup(expeval_variable_t *variable);

  /* Function symvar_resolve() resolves variable leaf nodes by locating
   * them in the supplied symbols table structure. If the
   * variable in the tree is resolved properly then 0 is returned.
   * Otherwise -1 is returned.
   */

int symvar_resolve(expeval_context_t *context, expeval_variable_t *variable)
   {
   expeval_node_t *child;        /* for iterating through child nodes */
   symbols_t *symbols;           /* symbols table structure from context */
   data_item_t *field;           /* identifier structure from symbols table */
   attribute_data_t *attributes; /* derived variable attribute data */
   int rc;

   if (!context) return -1;
   symbols = variable->resolve_private_data;
```

```
attributes = malloc(sizeof(*attributes));
memset(attributes,0,sizeof(*attributes));
variable->cleanup = cleanup;
variable->varaddress = symvar_address;
variable->private_data = attributes;

field = symbols_lookup_pathindex(symbols,variable->name,
      &attributes->levels,attributes->vector);
if (!field)
   {
   fprintf(stderr,"Unable to resolve variable: %s\n",variable->name);
   return -1;
   }

variable->varvalue = symvar_value;
variable->varvalue_private_data = field;
attributes->offset = symbols_index(field,attributes->levels,
      attributes->vector);
attributes->length = field->length;

/* If the status is left alone then this function will be called again
 * on priming the next expression that contains the variable.
 */
variable->status = EVAL_VAR_RESOLVED;

if (((field->category & CATEGORY_BITS) == NUMERIC)
   || ((field->category & CATEGORY_BITS) == (NUMERIC|SIGNED)))
   variable->value.type = EVAL_VALUE_NUMBER;
else
   variable->value.type = EVAL_VALUE_STRING;

return 0;
} /* symvar_resolve */
```

It is the responsibility of the resolve method to put in place all the functions and structures required to perform the extraction of the variable value in an efficient manner during subsequent expression evaluation processing.

5 Building Function Call Nodes

The process of building function nodes is similar to the process of building variable leaf nodes. Major differences are that the function call nodes need not be leaf nodes and in most cases would not be. This is because if the a function value were not dependent on the other values in the expression then the entity could be implemented as either a constant (in the case that the value is not dependent on anything during run-time) or the entity could be implemented as variable leaf node as the value can be determined from the dependants not known through evaluations of `expeval` expressions. Another major difference between variable leaf nodes and function nodes is that the methods

for implementing variable leaf nodes can be introduced as classes of variables (by the mechanism of associating the binding method to the first node of the variable), whereas for functions, each function is independently defined.

Functions are defined with the name of the entry point which defines the behaviour of the function. In addition, the function is defined with the type of the return value of the function and number and types of parameters to the function.

The following code registers the function for `substring`:

```
int substring(expeval_context_t *context,expeval_function_t *function,
            expeval_value_t *value, expeval_node_t *arguments);

   expeval_varsup_define_function(context,"substring",substring,cleanup,NULL,
         EVAL_VALUE_STRING,3,EVAL_VALUE_STRING,EVAL_VALUE_NUMBER,
         EVAL_VALUE_NUMBER);
```

Here the function `substring` is defined as taking three arguments, the first must be of type *String*, and the second and third of type *Number*. The defined function returns a value of type *String* and the function is implemented by the entry point `substring` (a C function).

A function call node is created using the `expeval_build_function` function which creates and returns an expression node. If the creation of the node fails then `NULL` is returned and a formatted error is placed in the `last_error` in the supplied context. Otherwise the created function expression node is returned.

The arguments of a function call node are supplied by the chain of expression nodes linked on the `peer` attributes of the expression nodes. In the following example, `ss4` is the resultant function call node and the argument chain is supplied by the expression node `ss1`:

```
   expeval_node_t *ss4;

   ss4 = expeval_build_function(context,"substring",ss1);
```

Like variable nodes, function reference nodes in an expression need to be resolved. The actual function is chosen when the function node is build and for this the function name is used a lookup. During expression typing (`expeval_type_expression`), which both checks and determines types, the types of the argument expressions is determined and then checked against the type stipulated when the function was defined. The resultant type is then the type of the supplied function return type. The type and number of arguments must match the function definition, otherwise the the typing fails and a message is placed in `last_error` of the supplied context.

During expression evaluation for function call nodes, the argument expressions are evaluated and if no error occurs then the function entry point that implements the function is called. It is the responsibility of the function to compute the required value from the supplied arguments and to insert the result into the result value.

```
int substring(expeval_context_t *context,expeval_function_t *function,
      expeval_value_t *value, expeval_node_t *arguments)
   {
   char *string;
   char *start;
   char *length;
   int s, l;

   string = expeval_value_to_string(context,&arguments->value);
   start = expeval_value_to_string(context,&arguments->peer->value);
   length = expeval_value_to_string(context,&arguments->peer->peer->value);

   sscanf(start,"%d",&s);
   sscanf(length,"%d",&l);
   if (value->string.string) free(value->string.string);
   value->string.length = l;
   value->string.string = malloc(value->string.length);
   memcpy(value->string.string,string+s-1,l);

   return 0;
   } /* substring */
```

Functions may acquire resources can be freed by supplying a `cleanup` procedure when they are registered (as above). When the `expeval` library cleans up a function definition, this cleanup function will be called just before the function structure is cleaned up. If no cleanup is required, then `NULL` can be passed as the cleanup entry point:

```
void cleanup(expeval_function_t *function)
   {
   printf("Cleaning up function %s.\n",function->name);
   } /* cleanup */
```

6 Composing Nodes into Expressions

To build an expression tree the various nodes need to be placed correctly in an expression evaluation tree. Notice that precedence is not an issue as the order of expression evaluation is determined by the tree structure.

The tree structure is obtained by populating the various `child` and `peer` members of the expression nodes. Assuming the following definitions, as they relate to the nodes of the expression tree in Figure 1, and that they have been created successfully:

```
   expeval_node_t *l1;          /* expression tree node for literal 1.3 */
   expeval_node_t *f;           /* expression tree node for function f */
   expeval_node_t *a;           /* expression tree node for variable a */
   expeval_node_t *b;           /* expression tree node for variable b */
   expeval_node_t *l2;          /* expression tree node for literal 4 */
   expeval_node_t *l3;          /* expression tree node for literal 10 */
```

The corresponding expression tree can be created by the population of the child-peer structure as follows:

```
/* Build the function call node. This takes as an argument the argument
 * chain to the function. This chain is put together first.
 */
 a->peer = b;
 b->peer = l2;
 f = expeval_build_function(context,"f",a);

/* Compose the full expression by joining all the other nodes into the
 * tree and building the interior operator nodes:
 */
 f->peer = l3;
 sub = expeval_build_interior(context,OP_MULTIPLY,f);

 l1->peer = sub;
 exp = expeval_build_interior(context,OP_ADD,l1);
```

This results in the expression tree of Figure 1 being rooted in the node `exp`.

Typically, the process of building an expression tree would be as a result the semantic actions of the reductions performed during a compile process. Function `expeval_compile()` compiles an expression tree from a text expression. The returned expression tree is an unprimed, untyped expression tree representing the given text.

```
expeval_node_t *expeval_compile(expeval_context_t *context,
      char *expression_string, expeval_output_callback_t callback,
      void *callback_parm);
```

The parameters `callback` and `callback_parm` provide a means of directing any output from the compiling process a user supplied function. The user supplied function is called with full line messages with the newline character. This function should have a prototype conforming to:

```
typedef void (*expeval_output_callback_t)(void *callback_parm, char *text);
```

And as an example, the following function satisfies this `typedef` and redirects the supplied messages to `stderr`:

```
static void default_callback(void *callback_parm, char *text)
   {
   fprintf(stderr,"%s\n",text);
   } /* default_callback */
```

The `callback` parameter may be supplied as `NULL` in which the output messages will be directed to `stderr` as in the snippet above.

Within the context of the previous examples, the following demonstrates an alternative method of deriving the same expression tree as the tree depicted in Figure 1 and rooted in `exp`. This time, though, the resultant expression tree will be rooted in the node `com`. If an error occurs during the parsing of the string, the the function will return `NULL` and

place an error message in the `last_error` field of the supplied context:

```
com = expeval_compile(context,"1.3+f(a,b,4)*10",NULL,NULL);
if (!com)
   {
   fprintf(stderr,"Error in string compile: %s\n",context->last_error);
   exit(16);
   }
```

The following program is a an example showing the entire process demonstrating the creation and evaluation of the expression from Figure 1:

```
void xbreakpoint(void) { return; }
  /* File: example.c
   *
   * This is a expeval library sample program which demonstrates the
   * preparation, building and execution of the expression:
   *
   * -a+1.3+f(a,b,4)*10
   *
   * This expression tree is built up in two ways. The first builds the
   * expression explicitly by calls and manipulation of the child parent
   * tree; and the second does so by compiling the expression tree from
   * a text string.
   *
   * Author: Stephen R. Donaldson [stephen@codemagus.com].
   *
   * Copyright (c) 2008 Code Magus Limited. All rights reserved.
   */

  /*
   * $Author: hayward $
   * $Date: 2023/09/25 17:02:29 $
   * $Id: example.c,v 1.64 2023/09/25 17:02:29 hayward Exp $
   * $Name:  $
   * $Revision: 1.64 $
   * $State: Exp $
   *
   * $Log: example.c,v $
   * Revision 1.64  2023/09/25 17:02:29  hayward
   * Add uuid time gen test
   * Allow program to continue if negative tests fail
   * Ask user if pstore is running before running those tests.
   *
   * Revision 1.63  2022/10/18 15:44:06  hayward
   * Allow a caller to de-register the values
   * callback function. This allows a caller
   * to register a values callback just before
   * calling expeval_evaluate() and then
   * de-registering it just after. The caller
   * can then decide for which evaluate calls
   * they need the call back active for or even
   * register and de-register different values
```

```
* call back routines for different evaluations
* on the same expeval context.
*
* Revision 1.62  2022/09/15 14:18:41  hayward
* Add the ability for a caller to register
* a variable value call back function. This
* then causes expeval to call the function
* for every variable processed in an expression
* within expeval_evaluate(). A user_data pointer
* and a name and value are passed to the routine.
* This allows a user to record the variables and
* their values that contributed to the result of
* an expression.
*
* Revision 1.61  2020/12/21 18:35:38  stephen
* Tidy up example.c - buffer overflow
*
* Revision 1.60  2020/12/20 12:17:03  stephen
* Add power function for integer powers
*
* Revision 1.59  2020/10/31 16:11:46  stephen
* Update example.c for random numbers and add check_random.R
*
* Revision 1.58  2020/10/31 14:15:28  stephen
* Add random number generator functions to expeval and documentation.
* Three random number generators are added which use numerical recipes
* as the underlying functions. The functions added are runif (for random
* uniformly distributed numbers); rnorm (for normally distributed random
* numbers); and rexp (for exponentially distributed random numbers).
*
* Revision 1.57  2020/10/27 08:46:04  hayward
* Reset test 4 to use derived values
* rather than literals.
*
* Revision 1.56  2020/10/24 16:02:51  stephen
* Correct expected type of numeric expression in example.c
*
* Revision 1.55  2020/10/24 15:59:53  stephen
* Change to use decdigits.h for value of DECNUMDIGITS
*
* Revision 1.54  2020/10/21 10:32:58  hayward
* Test sysstring returning 1E6 type numbers.
*
* Revision 1.53  2019/09/25 07:19:30  hayward
* Add SysStrTrimLeft and SysStrTrimRight.
*
* Revision 1.52  2018/05/16 12:03:10  stephen
* An funciton argument list can now be empty. This intrduces an
* OptExpressionList which is either empty or an ExpressionList.
*
* The function uuid_time_gen() has been added and the documentation
* updated. The documentation has been updated.
```

```
*
* The function uuid_time_gen() uses the libuuid library to generate
* a UUID which is expected to be unique and which is based on the
* clock of the local system and the MAC address of the local system's
* interface. It uses the uuid_generate_time() function from libuuid.
*
* Addtional changes and initialisation updates to guard against
* passing NULL expressions as a result of previous errors and now
* the global expression point that the parser updates is initialised
* for each compile (required in case the parser does not create
* an expression.
*
* Revision 1.51  2018/04/13 13:51:37  hayward
* t into 80bytes
*
* Revision 1.50  2018/04/13 13:47:02  hayward
* Add comment about unless cleanup abend.
* Email contents are
* Hi Stephen,
* I think that this is one for you as the abend is a double free in the
* depths of expeval_cleanup(). It is only found in the example.c program in
* expeval at the moment.
*
* It only occurs on a complex expression using unless.
* The expression is one I came up with to help Jan format a date in the MDM
* NFT that comes from DB2 and needs to be put in the XML.  It reformats the
* ISO date (excluding the microseconds) and then adds the microseconds if
* available. The invalid date had spaces in them.
*
* This is the expression
*
* SysStrCat(("1911-01-01 00:00:00." unless strftime(
*   SysTime(date,"%Y-%m-%d-%H.%M.%S"),"%Y-%m-%d %H:%M:%S."),
*   (ifelse(SysSubStr(date,1,1)=" ","000000",SysSubStr(date,21,6))))
*
* If I change it to
*
* SysStrCat(                                strftime(
*   SysTime(date,"%Y-%m-%d-%H.%M.%S"),"%Y-%m-%d %H:%M:%S." ),
*   (ifelse(SysSubStr(date,1,1)=" ","000000",SysSubStr(date,21,6))))
*
* there is no abend.
*
* Do you want me to set up a test core dump and or gdb run for you at some
* time?
*
* The code is
*    strncpy(date_varvalue,"2017-07-15-13.07.56.000000",
*              sizeof(date_varvalue));
*    rc = expeval_evaluate(context,exp);
*    if (rc) error_exit(context,NULL);
*    printf("date=\"%s\"\n",date_varvalue);
```

```
*    printf("%s ===> %s\n",exp_text,value_to_string(context,&exp->value));
*    rc = expeval_final(context,exp);
*    if (rc) error_exit(context,NULL);
*
*    expeval_cleanup(exp);                    <----- Abends in here
*    expeval_context_close(context);
*
*
* Regards
* Patrick
*
* Revision 1.49  2018/04/13 13:40:07  hayward
* Add a complex unless and date expression.
* Implement incremental test number and
* start of test messages.
*
* Revision 1.48  2017/12/04 16:17:52  hayward
* Make pstore_set, pstore_get and pstore_get_cset
* all return a type string and if applicable
* accept a value of type string. This makes them
* more generic. Only pstore_get_incr and
* pstore_get_incr_cset return and accept numeric
* values.
*
* Revision 1.47  2017/12/04 12:31:02  hayward
* Add functionality that enables
* getting, setting and incrementing
* the value of a numeric persistent
* store variable. This change will
* require changes to the link of
* every program that uses expeval.
*
* Revision 1.46  2017/11/17 17:31:37  hayward
* Better control over error handling
* Add test of expression with multiple lookups
* which should all use the same lookup hash
* table internally.
* Add isalias() or inlookup() test.
*
* Revision 1.45  2017/11/16 14:06:11  hayward
* Improve diagnostic messages.
*
* Revision 1.44  2017/11/06 16:45:02  hayward
* Update documentation and examples
* with new built in functions
* sfget, sfset, gsub and lookup.
*
* Revision 1.43  2017/11/04 17:26:33  stephen
* Signed numbers not recogbized by lexical analyser. Rather parser
* treats sign as unary operator.
*
* Revision 1.42  2017/11/03 15:09:03  hayward
```

```
* Change variable with hyphens,
* which was a test, to one with
* underscores.
*
* Revision 1.41  2017/11/02 23:22:30  hayward
* Cosmetic changes only.
*
* Revision 1.40  2017/10/30 15:36:48  hayward
* Add alias() (or lookup()) function.
* This allows the substiturion of one value
* for another. Look up table is supplied as
* keyword=value comma separated pairs or
* line separated if in a file.
*
* Revision 1.39  2017/10/24 22:16:11  hayward
* Add failing gensub check.
*
* Revision 1.38  2017/10/24 22:12:30  hayward
* Add gsub (alias replace) to expeval.
*
* Revision 1.37  2017/08/19 10:00:49  stephen
* Add integer division operator div to expeval
*
* Revision 1.36  2015/10/26 11:42:22  stephen
* Add support for condition packing of string from display hex back to
* local character encoding and allow the replacement of non-graphic
* or non-printable characters by a supplied or defaulted character.
*
* Revision 1.35  2015/07/09 17:31:49  hayward
* Change method for DE48 Structured Data
* field extraction.
* Add DE48 structured field update routine.
* Add tests for both DE48 get and set routines.
*
* Revision 1.34  2015/07/03 07:27:04  hayward
* Add function to extract a name-value pair
* from the TermApp DE48 structured data field.
* From the spec this field is made up of consecutive
* fields laid out as follows:
* . 1 byte length indicator of the key length indicator
* . Length indicator of the key
* . Key
* . 1 byte length indicator of the value length indicator
* . Length indicator of the value
* . Value
*
* Revision 1.33  2015/03/19 20:07:57  stephen
* Allow strings to be empty/zero length and add builtin strlen function
*
* Revision 1.32  2015/01/22 09:41:38  hayward
* Changes for MVS.
* Remove popt.
```

```
* Time format %s for microseconds is not supported.
*
* Revision 1.31  2014/12/09 23:58:21  stephen
* Add ifelse-operator and update documentation
*
* Revision 1.30  2014/12/09 19:49:37  stephen
* Add C/awk alias names for predefined funcitons and add strftime
*
* Revision 1.29  2014/12/05 17:21:57  stephen
* Add test case for Boolean result
*
* Revision 1.28  2014/10/22 15:32:31  hayward
* Only run the big expression test
* if asked to do so specifically.
*
* Revision 1.27  2014/09/09 17:16:34  stephen
* Add conditional unless-operator
*
* Revision 1.26  2014/07/22 17:14:26  hayward
* Add SysStrPadLeft() and SysStrPadRight()
*
* Revision 1.25  2014/06/23 21:52:15  hayward
* Add SysTime()
*
* Revision 1.24  2014/06/11 09:31:13  hayward
* Add system functions for strspn() and strcspn().
*
* Revision 1.23  2013/02/12 09:17:11  hayward
* Add popt options that allow the
* program to continue on an error.
*
* Revision 1.22  2012/07/24 09:48:44  hayward
* Add Large File Support (LFS)
*
* Revision 1.21  2012/02/21 16:51:09  stephen
* example updates for unique
*
* Revision 1.20  2011/10/22 13:59:01  stephen
* Fix for unique operator, add example and documentation
*
* Revision 1.19  2011/10/22 12:42:38  stephen
* Update to sample program for new test case
*
* Revision 1.18  2011/08/20 20:22:52  stephen
* Add function SysStrStr for return pos arg1 in arg2
*
* Revision 1.17  2011/07/31 13:00:44  stephen
* Add support for unique operator
*
* Revision 1.16  2010/09/22 15:03:02  hayward
* Move the callback and callback parameter
* to the context_open function from the
```

```
* compile function and make the variables
* part of the context structure. This allows
* both the caller and the expeval library to
* access these values consistently.
*
* Revision 1.15  2009/11/02 11:35:05  stephen
* Add direct method for evaluating arithmetic negation
*
* Revision 1.14  2009/11/02 09:39:22  stephen
* Add warning regarding using pointers in common nodes
*
* Revision 1.13  2009/10/06 21:08:03  stephen
* Add pre-defined functions SysStrCat, SysString, SysNumber and
* SysFmtCurrTime
*
* Revision 1.12  2009/10/06 19:22:02  stephen
* Add Support for Builtin Function SysSubStr
*
* Revision 1.11  2008/10/21 12:22:09  hayward
* Add version and usage to test programs.
*
* Revision 1.10  2008/05/29 00:45:13  stephen
* Fix type of sample enum field list function
*
* Revision 1.9  2008/05/28 15:48:36  stephen
* Enumerate available fields in an objtype in example
*
* Revision 1.8  2008/05/28 15:10:29  stephen
* Update the example showing objtypes evaluation of expressions
*
* Revision 1.7  2008/05/28 13:11:39  stephen
* Objvar testing in expressions.
*
* Revision 1.6  2008/05/28 11:53:55  stephen
* Changes to deal with failed expression compilation and error messages
*
* Revision 1.5  2008/05/21 09:36:24  stephen
* Cleanup example program expression cleanup
*
* Revision 1.4  2008/04/14 13:10:56  stephen
* Changes for Windows build
*
* Revision 1.3  2008/04/08 17:58:35  stephen
* Fixes required comming out of integration into objtypes
*
* Revision 1.2  2008/04/07 19:44:36  stephen
* Add support for compiling strings into expression trees
*
* Revision 1.1  2008/01/23 19:15:36  stephen
* Add documentation and fix variable return value
*
*/
```

```
static char *sample_c_cvs =
   "$Id: example.c,v 1.64 2023/09/25 17:02:29 hayward Exp $";

  /*
   * Large File Support Required from various environments:
   */

#ifndef __HOS_MVS__
#define _LARGEFILE_SOURCE
#define _LARGEFILE64_SOURCE
#define _LARGE_FILES
#define _FILE_OFFSET_BITS 64
#endif

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#ifndef __HOS_MVS__
#include <popt.h>
#endif
#include <stdarg.h>

#include <expeval.h>

#include <objtypes.h>
#include <objvar.h>
#include "version.h"

  /*
   * Globals.
   */

   int tstnum = 0;

  /*
   * Command line parsing popt data structure and option variables.
   */

int force = 0;                      /* Do not stop on an error */
int prompt = 0;                     /* Prompt for action on an error */
int skip_pstore = 0;                /* Skip pstore tests */
int big_expression = 0;             /* Also run the big expression which may
                                       take a long time */
int verbose = 0;                    /* verbose processing selected */

#ifndef __HOS_MVS__
struct poptOption optionsTable[] =
   {
      {  "force", 'f', POPT_ARG_NONE, &force, 0,
         "Do not stop on an error", NULL},
      {  "prompt", 'p', POPT_ARG_NONE, &prompt, 0,
```

```
         "Prompt for action on an error", NULL},
      {  "skip-pstore", 's', POPT_ARG_NONE, &skip_pstore, 0,
         "Do not run pstore tests", NULL},
      {  "big-expression", 'b', POPT_ARG_NONE, &big_expression, 0,
         "Also test a very big expression; May take a while", NULL},
      {  "verbose", 'v', POPT_ARG_NONE, &verbose, 0,
         "Verbose processing mode", NULL},
      POPT_AUTOHELP
      { NULL, 0, 0, NULL, 0 }
   };
#endif

  /*
   * Local prototypes:
   */

static char *value_to_string(expeval_context_t *context,
      expeval_value_t *value);
static void error_exit(expeval_context_t *context, char *message);
static int resolve(expeval_context_t *context, expeval_variable_t *variable);
static int varvalue(expeval_context_t *context, expeval_variable_t *variable);
static int funentry(expeval_context_t *context,expeval_function_t *function,
            expeval_value_t *value, expeval_node_t *arguments);
int list_fields(objtype_t *objtype, objfield_t *objfield, void *callback_parm);
static int resolve_string(expeval_context_t *context,
      expeval_variable_t *variable);
static int varstring(expeval_context_t *context, expeval_variable_t *variable);
static int lprintf(char *format, ...);
static void VC_callback(void *user_data, char *keyword, char *value);
static char *FROM(int l);

int main(int argc, char **argv)
   {
#ifndef __HOS_MVS__
   poptContext optCon;           /* for processing command line arguments */
#endif
   expeval_context_t *context;   /* context for all expeval functions */
   expeval_node_t *exp;          /* final expression root */
   expeval_node_t *sub;          /* intermediate sub-expression node */
   expeval_node_t *l1;           /* expression tree node for literal 1.3 */
   expeval_node_t *f;            /* expression tree node for functon f */
   expeval_node_t *a;            /* expression tree node for variable a */
   expeval_node_t *b;            /* expression tree node for variable b */
   expeval_node_t *l2;           /* expression tree node for literal 4 */
   expeval_node_t *l3;           /* expression tree node for literal 10 */
   expeval_supplier_t *varsup;   /* supplier of variable values */
   int va;                       /* underlying value of variable a */
   int vb;                       /* underlying value of variable b */
   expeval_value_t value;        /* literal values to place into nodes */
   expeval_node_t *com;          /* final expression root from compile */
   char *type_name;              /* type name of objtypes. */
   objtypes_t *types;            /* object types collection structure */
```

```
   objtype_t *type;              /* object type structure */
   char exp_text[32760];         /* expressoin in text form */
   char *big_text;               /* for compiling very big expressions */
   unsigned char buffer[23];     /* buffer for evaluating expression */
   int buflen;                   /* length of content in buffer */
   char date_varname[100];       /* Date variable name */
   char date_varvalue[100];      /* Date variable value */
   char gsub_varname[10][100];   /* Test regex variable name */
   char gsub_varvalue[10][100];  /* Test regex variable value */
   char alias_varname[100];      /* Alias lookup variable */
   char alias_varname2[100];     /* Alias lookup variable */
   char alias_varname3[100];     /* Alias lookup variable */
   char alias_varvalue[100];     /* Alias looked up return variable */
   char alias_varvalue2[100];    /* Alias looked up return variable */
   char alias_varvalue3[100];    /* Alias looked up return variable */
   expeval_node_t *gsub_node[10];/* To resolve regex */
   expeval_supplier_t *gsub_varsup; /* supplier of variable values */
   char pstore_varname[64];      /* pstore variable name */
   char pstore_varvalue[64];     /* pstore variable tolue */
   char pstore_hostname[64];     /* pstore host:port name */
   char pstore_hostvalue[64];    /* pstore host:port value */
   char pstore_svalname[64];     /* pstore string value name */
   char pstore_svalvalue[64];    /* pstore string value value */
   char pstore_nvalname[64];     /* pstore numeric value name */
   char VC_varname[10][100];     /* Test value callback */
   char VC_varvalueS[5][100];    /* Test value callback String */
   int  VC_varvalueI[5];         /* Test value callback integer */
   char *VC_user_data = "VC";    /* User data for VC callback test */
   int pstore_nvalvalue;         /* pstore numeric value value */
   FILE *csvfile;                /* for random number sampling */
   char *tmp;
   int i,j;
   int rc;

  /* Process the command line options and populate the command line option
   * variables.
   */
#ifndef __HOS_MVS__
   optCon = poptGetContext(argv[0],argc,(const char **)argv,optionsTable,0);
   rc = poptGetNextOpt(optCon);
   if (rc < -1)
      {
      fprintf(stderr,"%s\n",poptStrerror(rc));
      poptPrintUsage(optCon, stderr, 0);
      exit(16);
      }
#endif

  /* Create a context in which all expeval library calls can be made.
   */
   context = expeval_context_open(0,NULL,NULL);
```

```
/* Define two variable suppliers. One for each of the variable variables
 * in the expression.
 */
 varsup = expeval_varsup_open(context,"a",resolve,NULL,varvalue,&va);
 varsup = expeval_varsup_open(context,"b",resolve,NULL,varvalue,&vb);

/* Introduce the function to the expeval library:
 */
 expeval_varsup_define_function(context,"f",funentry,NULL,&va,
       EVAL_VALUE_NUMBER,
       3,EVAL_VALUE_NUMBER,EVAL_VALUE_NUMBER,EVAL_VALUE_NUMBER);

/* Build the literals required by the expression:
 */
 rc = expeval_value_number(context,"1.3",&value);
 if (rc) error_exit(context,FROM(__LINE__));

 l1 = expeval_build_lit_leaf(context,&value);
 if (!l1) error_exit(context,FROM(__LINE__));

 rc = expeval_value_number(context,"4",&value);
 if (rc) error_exit(context,FROM(__LINE__));

 l2 = expeval_build_lit_leaf(context,&value);
 if (!l2) error_exit(context,FROM(__LINE__));

 rc = expeval_value_number(context,"10",&value);
 if (rc) error_exit(context,FROM(__LINE__));

 l3 = expeval_build_lit_leaf(context,&value);
 if (!l3) error_exit(context,FROM(__LINE__));


/* Build the variable reference nodes:
 */
 a = expeval_build_var_leaf(context,strdup("a"));
 if (!a) error_exit(context,FROM(__LINE__));

 b = expeval_build_var_leaf(context,strdup("b"));
 if (!b) error_exit(context,FROM(__LINE__));

/* Buid the function call node. This takes as an argument the argument
 * chain to the function. This chain is put together first.
 */
 a->peer = b;
 b->peer = l2;
 f = expeval_build_function(context,"f",a);
 if (!f) error_exit(context,FROM(__LINE__));

/* Compose the full expression by joining all the other nodes into the
 * tree and building the interior operator nodes:
 */
```

```
   f->peer = l3;
   sub = expeval_build_interior(context,OP_MULTIPLY,f);

   l1->peer = sub;
   exp = expeval_build_interior(context,OP_ADD,l1);
   if (!exp) error_exit(context,FROM(__LINE__));

  /* Make sure that the expression types correctly and propagate the
   * typing up to the top root:
   */
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

  /* Prepare the expression for execution:
   */
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

  /* Evaluate the expression:
   */
   va = 15;
   vb = 25;
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

  /* Show the expression evaluation result:
   */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"Basic expressions");
   lprintf("Answer to expression evaluation when a = %d and b = %d.\n",va,vb);
   lprintf("1.3+f(a,b,4)*10 = %s\n",
         value_to_string(context,&exp->value));
   lprintf("Correct answer  = %f\n",1.3+(va+vb+4)*10);

  /* Cleanup the expression, but leave in a state that the expression
   * tree can be reused.
   */
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

/* Compile arithmetic expression */
  /* Now build the expression tree from a text string which contains the
   * expression in text form.
   */
   com = expeval_compile(context,"1.3+f(a,b,4)*10");
   if (!com) error_exit(context,FROM(__LINE__));

  /* Make sure that the expression types correctly and propagate the
   * typing up to the top root:
   */
   rc = expeval_type_expression(context,com);
   if (rc) error_exit(context,FROM(__LINE__));
```

```
/* Prepare the expression for execution:
 */
 rc = expeval_prime(context,com);
 if (rc) error_exit(context,FROM(__LINE__));

/* Evaluate the expression:
 */
 rc = expeval_evaluate(context,com);
 if (rc) error_exit(context,FROM(__LINE__));

/* Show the expression evaluation result:
 */
 lprintf("Answer to expression evaluation when a = %d and b = %d.\n",va,vb);
 lprintf("-a+1.3+f(a,b,4)*10 = %s\n",
       value_to_string(context,&com->value));
 lprintf("Correct answer  = %f\n",-va+1.3+(va+vb+4)*10);

/* Evaluate the expression, but with different values of a and b.
 */
 va = 150;
 vb = 250;
 rc = expeval_evaluate(context,com);
 if (rc) error_exit(context,FROM(__LINE__));

/* Show the expression evaluation result:
 */
 lprintf("Answer to expression evaluation when a = %d and b = %d.\n",va,vb);
 lprintf("-a+1.3+f(a,b,4)*10 = %s\n",
       value_to_string(context,&com->value));
 lprintf("Correct answer  = %f\n",-va+1.3+(va+vb+4)*10);

/* Cleanup the expression, but leave in a state that the expression
 * tree can be reused.
 */
 rc = expeval_final(context,com);
 if (rc) error_exit(context,FROM(__LINE__));

/* Cleanup the expression and the context and free resources acquired
 * during processing:
 */
 expeval_cleanup(com);
 expeval_cleanup(exp);

/* Test the unique operator:
 */
 exp = expeval_compile(context,"unique (a+b)");
 if (!exp) error_exit(context,FROM(__LINE__));

/* Make sure that the expression types correctly and propagate the
 * typing up to the top root:
 */
 rc = expeval_type_expression(context,exp);
```

```
 if (rc) error_exit(context,FROM(__LINE__));

/* Prepare the expression for execution:
 */
 rc = expeval_prime(context,exp);
 if (rc) error_exit(context,FROM(__LINE__));

/* Evaluate various unique expression:
 */
 va = 1;
 vb = 1;

 rc = expeval_evaluate(context,exp);
 if (rc) error_exit(context,FROM(__LINE__));

 lprintf(" a = %d, b = %d: ",va,vb);
 lprintf("unique a+b = %s\n",value_to_string(context,&exp->value));

 va = 1;
 vb = 1;

 rc = expeval_evaluate(context,exp);
 if (rc) error_exit(context,FROM(__LINE__));

 lprintf(" a = %d, b = %d: ",va,vb);
 lprintf("unique a+b = %s\n",value_to_string(context,&exp->value));

 va = 2;
 vb = 1;

 rc = expeval_evaluate(context,exp);
 if (rc) error_exit(context,FROM(__LINE__));

 lprintf(" a = %d, b = %d: ",va,vb);
 lprintf("unique a+b = %s\n",value_to_string(context,&exp->value));

 va = 3;
 vb = 2;

 rc = expeval_evaluate(context,exp);
 if (rc) error_exit(context,FROM(__LINE__));

 lprintf(" a = %d, b = %d: ",va,vb);
 lprintf("unique a+b = %s\n",value_to_string(context,&exp->value));

 va = 2;
 vb = 3;

 rc = expeval_evaluate(context,exp);
 if (rc) error_exit(context,FROM(__LINE__));

 lprintf(" a = %d, b = %d: ",va,vb);
```

```
   lprintf("unique a+b = %s\n",value_to_string(context,&exp->value));

  /* Cleanup:
   */
   expeval_cleanup(exp);
   expeval_context_close(context);

  /* Check a failure situation:
   */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"Failure 1");
   context = expeval_context_open(0,NULL,NULL);

   strcpy(exp_text,"record1.cover2.field3 >= 10.75");
   com = expeval_compile(context,exp_text);
   if (com) lprintf("Compile should not return a node --- bad!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- good!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   expeval_cleanup(com);
   expeval_context_close(context);

  /* This one should work:
   */
#ifndef __HOS_MVS__
   type_name = "testtype.objtypes";
#else
   type_name = "DD:TESTTYPE";
#endif
   types = objtypes_open(type_name,0);
   if (!types)
      {
      fprintf(stderr,"Failed to create object types structure from %s\n",
         type_name);
      exit(1);
      }
   type = objtypes_lookup_type(types,"record");
   lprintf("Start of test %3.3d: %s\n",++tstnum,"Basic objtypes");
   context = expeval_context_open(0,NULL,NULL);
   varsup = expeval_varsup_open(context,type->name,
         objvar_resolve,NULL,objvar_value,type);
   strcpy(exp_text,"record.covera.sectionb.field3 >= 10.75");
   com = expeval_compile(context,exp_text);
   if (com) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   expeval_cleanup(com);
   expeval_context_close(context);
```

```
 objtypes_close(types);

/* This one should work:
 */
 types = objtypes_open(type_name,0);
 if (!types)
    {
    fprintf(stderr,"Failed to create object types structure (2) from %s\n",
       type_name);
    exit(1);
    }
 type = objtypes_lookup_type(types,"record");
 objtypes_enum_fieldlist(type,list_fields,NULL);
 lprintf("Start of test %3.3d: %s\n",++tstnum,"Basic objtypes 3");
 context = expeval_context_open(0,NULL,NULL);
 context->default_var_supplier = expeval_varsup_open(context,"default supp",
       objvar_resolve,NULL,objvar_value,type);

 strcpy(exp_text,"(covera.sectionb.field4 / 100.00)");
 exp = expeval_compile(context,exp_text);
 if (exp) lprintf("Compile should return a node --- good!\n");
 else
    {
    lprintf("Invalid compile did not return a node --- bad!\n");
    lprintf("Last error: %s\n",context->last_error);
    }

 rc = expeval_type_expression(context,exp);
 if (rc) error_exit(context,FROM(__LINE__));
 if (exp->value.type != EVAL_VALUE_NUMBER)
    lprintf("The expression %s is expected to be numeric.\n",exp_text);
 rc = expeval_prime(context,exp);
 if (rc) error_exit(context,FROM(__LINE__));

/* From here on would be in a loop if the expression is being repeatedly
 * evaluated over a record set:
 */
 /* First time */
 memcpy(buffer,"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x0A"
       "\x30\x30\x30\x30\x30\x30\x30\x31\x30\x30\x30\x30",23);
 buflen = 23;
 context->buf = buffer;
 context->buf_len = buflen;
 rc = expeval_evaluate(context,exp);
 if (rc) error_exit(context,FROM(__LINE__));
 lprintf("%s = %s\n",exp_text,value_to_string(context,&exp->value));
 /* Second time */
 memcpy(buffer,"\x41\x41\x41\x41\x41\x41\x00\x00\x00\x00\x9A"
       "\x30\x30\x30\x30\x30\x31\x32\x33\x34\x35\x30\x30",23);
 buflen = 23;
 context->buf = buffer;
 context->buf_len = buflen;
```

```
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("%s = %s\n",exp_text,value_to_string(context,&exp->value));
   /* From here would be outside the loop of the record set */
   /* Once the loop has ended then cleanup for the expression is done */

   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);
   objtypes_close(types);

  /* This one should work:
   */
#ifndef __HOS_MVS__
   type_name = "testtype.objtypes";
#else
   type_name = "DD:TESTTYPE";
#endif
   types = objtypes_open(type_name,0);
   if (!types)
      {
      fprintf(stderr,"Failed to create object types structure from %s\n",
         type_name);
      exit(1);
      }
   type = objtypes_lookup_type(types,"record");
   lprintf("Start of test %3.3d: %s\n",++tstnum,"Basic objtypes");
   context = expeval_context_open(0,NULL,NULL);
   varsup = expeval_varsup_open(context,type->name,
         objvar_resolve,NULL,objvar_value,type);
   strcpy(exp_text,"record.covera.sectionb.field3 >= 10.75");
   com = expeval_compile(context,exp_text);
   if (com) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   expeval_cleanup(com);
   expeval_context_close(context);
   objtypes_close(types);

  /* This one should work:
   */
   types = objtypes_open(type_name,0);
   if (!types)
      {
      fprintf(stderr,"Failed to create object types structure (2) from %s\n",
         type_name);
      exit(1);
      }
```

```
   type = objtypes_lookup_type(types,"record");
   objtypes_enum_fieldlist(type,list_fields,NULL);
   lprintf("Start of test %3.3d: %s\n",++tstnum,"Basic objtypes 2");
   context = expeval_context_open(0,NULL,NULL);
   context->default_var_supplier = expeval_varsup_open(context,"default supp",
         objvar_resolve,NULL,objvar_value,type);

   strcpy(exp_text,"covera.sectionb.field3 >= 10.75");
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }

   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_BOOLEAN)
      lprintf("The expression %s is expected to be boolean.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

  /* From here on would be in a loop if the expression is being repeatedly
   * evaluated over a record set:
   */
   /* First time */
   memcpy(buffer,"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x0A",11);
   buflen = 11;
   context->buf = buffer;
   context->buf_len = buflen;
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("%s = %s\n",exp_text,value_to_string(context,&exp->value));
   /* Second time */
   memcpy(buffer,"\x41\x41\x41\x41\x41\x41\x00\x00\x00\x00\x9A",11);
   buflen = 11;
   context->buf = buffer;
   context->buf_len = buflen;
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("%s = %s\n",exp_text,value_to_string(context,&exp->value));
   /* From here would be outside the loop of the record set */
   /* Once the loop has ended then cleanup for the expression is done */

   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);
   objtypes_close(types);

/*Boolean Result*/
```

```
   strcpy(exp_text,"1=1");
   lprintf("Checking [%s] = [True];\n",exp_text);
   lprintf("Start of test %3.3d: %s\n",++tstnum,"Boolean");
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_BOOLEAN)
      lprintf("The expression %s is expected to be boolean.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*Power Result*/
   strcpy(exp_text,"1.41421356237309504880^2");
   lprintf("Checking [%s] = [2];\n",exp_text);
   lprintf("Start of test %3.3d: %s\n",++tstnum,"Power");
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be numeric.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/* Random numbers runif(200,300) */
```

```
   strcpy(exp_text,"runif(200,300)");
   lprintf("Checking [%s] distribution\n",exp_text);
   lprintf("Start of test %3.3d: %s\n",++tstnum,"runif");
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be numeric.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   csvfile = fopen("/tmp/runif.csv","w");
   fprintf(csvfile,"RUNIF\n");
   for (i = 0; i < 1000000; i++)
      {
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      fprintf(csvfile,"%s\n",value_to_string(context,&exp->value));
      }
   fclose(csvfile);
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/* Random numbers rnorm(1000,100) */
   strcpy(exp_text,"rnorm(1000,100)");
   lprintf("Checking [%s] distribution\n",exp_text);
   lprintf("Start of test %3.3d: %s\n",++tstnum,"rnorm");
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be numeric.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   csvfile = fopen("/tmp/rnorm.csv","w");
   fprintf(csvfile,"RNORM\n");
   for (i = 0; i < 1000000; i++)
```

```
      {
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      fprintf(csvfile,"%s\n",value_to_string(context,&exp->value));
      }
   fclose(csvfile);
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/* Random numbers rexp(10) */
   strcpy(exp_text,"rexp(10)");
   lprintf("Checking [%s] distribution\n",exp_text);
   lprintf("Start of test %3.3d: %s\n",++tstnum,"rexp");
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be numeric.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   csvfile = fopen("/tmp/rexp.csv","w");
   fprintf(csvfile,"REXP\n");
   for (i = 0; i < 1000000; i++)
      {
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      fprintf(csvfile,"%s\n",value_to_string(context,&exp->value));
      }
   fclose(csvfile);
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/* Random numbers 105.562+rexp(25.11205) */
   strcpy(exp_text,"105.562+rexp(25.11205)");
   lprintf("Checking [%s] distribution\n",exp_text);
   lprintf("Start of test %3.3d: %s\n",++tstnum,"rexp");
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
```

```
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be numeric.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   csvfile = fopen("/tmp/rexp.csv","w");
   fprintf(csvfile,"REXP\n");
   for (i = 0; i < 1000000; i++)
      {
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      fprintf(csvfile,"%s\n",value_to_string(context,&exp->value));
      }
   fclose(csvfile);
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);
/*String from number Result*/
   strcpy(exp_text,"string(1234567890+3.14159265358979311599 8*3.14159265358979311599 8
   strcpy(exp_text,"000000010000/100");
   lprintf("Checking [%s] = [3.14159265358979311599 8];\n",exp_text);
   lprintf("Start of test %3.3d: %s\n",++tstnum,"String from number");
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

  /* Test Built in functions:
   */
/*strlen()*/
```

```
   lprintf("Start of test %3.3d: %s\n",++tstnum,"strlen");
   strcpy(exp_text,"strlen(\"\")");
   lprintf("Checking [%s] = [0];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be numeric.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*strlen()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"strlen 2");
   strcpy(exp_text,"strlen(\"12345678\")");
   lprintf("Checking [%s] = [8];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be numeric.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*strlen()*/
```

```
   lprintf("Start of test %3.3d: %s\n",++tstnum,"strlen 3");
   strcpy(exp_text,"strlen(substr(\"12345678\",1,0))");
   lprintf("Checking [%s] = [0];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be numeric.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysSubStr()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysSubStr");
   strcpy(exp_text,"SysSubStr(\"abcdefghi\",4,3) = \"def\"");
   lprintf("Checking [%s] = [1];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_BOOLEAN)
      lprintf("The expression %s is expected to be boolean.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysSubStr()*/
```

```
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysSubStr 2");
/* strcpy(exp_text,"SysSubStr(\"abcdefghi\",0,3) = \"abcdefghi\""); */
   strcpy(exp_text,"SysSubStr(\"abcdefghi\",1,3) = \"abcdefghi\"");
   lprintf("Checking [%s] = [1];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_BOOLEAN)
      lprintf("The expression %s is expected to be boolean.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysString()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysString");
   strcpy(exp_text,"SysString(123) = \"123\"");
   lprintf("Checking [%s] = [1];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_BOOLEAN)
      lprintf("The expression %s is expected to be boolean.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);
```

```
/*SysNumber()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysNumber");
   strcpy(exp_text,"SysNumber(\"897\") = 897");
   lprintf("Checking [%s] = [1];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_BOOLEAN)
      lprintf("The expression %s is expected to be boolean.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysStrCat()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrCat");
   strcpy(exp_text,"SysStrCat(\"ABC\",\"DEF\") = \"ABCDEF\"");
   lprintf("Checking [%s] = [1];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_BOOLEAN)
      lprintf("The expression %s is expected to be boolean.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);
```

```
/*SysFmtCurrTime()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysFmtCurrTime");
   strcpy(exp_text,"SysFmtCurrTime(\"%m%y\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysTime*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysTime");
   strcpy(exp_text,"SysTime(\"2014-03-02 12:34:56\",\"%Y-%m-%d %H:%M:%S\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be numeric.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);
```

```
/*strftime() - MVS (and maybe other platforms) does not support %s for
 * microseconds - so for this test we exclude them by using SysSubStr() */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"strftime");
   strcpy(exp_text,
       "strftime(SysTime(SysSubStr(\"2017-07-15-13.07.56.413000\",1,21),"
       "\"%Y-%m-%d-%H.%M.%S\"),\"%Y-%m-%d %H:%M:%S\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*strftime() - not on MVS
 * example of bad formatting, use "unless" to provide a default date if the
 * formatting fails. The "a" in the date causes SysTime and strftime to fail.
 */
   xbreakpoint();
#ifndef __HOS_MVS__
   lprintf("Start of test %3.3d: %s\n",++tstnum,"Complex date");

   context = expeval_context_open(0,NULL,NULL);
   lprintf("Compiling [%s];\n",exp_text);

   strncpy(date_varname,"date",sizeof(date_varname));

   lprintf("expeval_varsup_open %s => value @%p\n",date_varname,date_varvalue);
   varsup = expeval_varsup_open(context,date_varname,resolve_string,NULL,
         varstring,&date_varvalue);
   if (!varsup) error_exit(context,FROM(__LINE__));

   strcpy(exp_text,
       "SysStrCat((\"1911-01-01 00:00:00.\" unless "
       "strftime(SysTime(date,"
       "\"%Y-%m-%d-%H.%M.%S\"),\"%Y-%m-%d %H:%M:%S.\")),"
       "(ifelse(SysSubStr(date,1,1)=\" \",\"000000\","
```

```
       "SysSubStr(date,21,6))))");

   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

   strncpy(date_varvalue,"                          ",sizeof(date_varvalue));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("date=\"%s\"\n",date_varvalue);
   lprintf("%s ===> %s\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

   strncpy(date_varvalue,"2017-07-15-13.07.56.413000",sizeof(date_varvalue));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("date=\"%s\"\n",date_varvalue);
   lprintf("%s ===> %s\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

   strncpy(date_varvalue,"2017-07-15-13.07.56.000000",sizeof(date_varvalue));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("date=\"%s\"\n",date_varvalue);
   lprintf("%s ===> %s\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

  /* On CENTOS64 the following causes a double free. If the unless clause is
   * removed from the above expression then the expeval_cleanup() works fine.
   * I have left the statement commented out until it is resolved in expeval. A
   * note has been sent to SD. Email title is
   * "expeval abend in expeval_cleanup(expression) after using unless."
   */
#if 0
   expeval_cleanup(exp);
#endif
   expeval_context_close(context);
#endif
```

```
/*SysStrStr()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrStr");
   strcpy(exp_text,"SysStrStr(\"cat\",\"Is the cat amongst the pigieons\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be number.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysStrStr()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrStr 2");
   strcpy(exp_text,"SysStrStr(\"dog\",\"Is the cat amongst the pigieons\")");
   strcpy(exp_text,"SysSubStr(\"1234567890123456=bla bla\",1,"
         "-1+SysStrStr(\"=\",\"1234567890123456=bla bla\"))");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be number.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
```

```
   expeval_context_close(context);

/*SysStrSpn()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrSpn");
   strcpy(exp_text,"SysStrSpn(\"1234567890123456=0116blabla\",\"0123456789\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be number.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrSpn 2");
   strcpy(exp_text,"SysStrSpn(\"1234567890123456D0116blabla\",\"0123456789\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be number.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);
```

```
/*SysStrCspn()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrCspn");
   strcpy(exp_text,"SysStrCspn(\"1234567890123456=0116bla bla\",\"=D\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be number.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrCspn 2");
   strcpy(exp_text,"SysStrCspn(\"1234567890123456D0116bla bla\",\"=D\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be number.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);
```

```
/*SysTime()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysTime");
   strcpy(exp_text,"SysTime(\"2014-03-02 12:34:56\",\"%Y-%m-%d %H:%M:%S\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be number.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysTime()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysTime 2");
   strcpy(exp_text,"SysTime(\"1970-01-01 01:00:00\",\"%Y-%m-%d %H:%M:%S\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be number.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);
```

```
/*SysTime()  %s is not defined on MVS */
#ifndef __HOS_MVS__
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysTime 3");
   strcpy(exp_text,"SysTime(\"4294967295\",\"%s\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be number.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);
#endif

/*SysFmtCurrTime() and SysTime()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysTime SysFmtCurrTime");
   strcpy(exp_text,"SysTime("
         "SysFmtCurrTime(\"%Y-%m-%d %H:%M:%S\"),\"%Y-%m-%d %H:%M:%S\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
```

```
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysFmtCurrTime()  %s is not defined on MVS */
#ifndef __HOS_MVS__
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysFmtCurrTime");
   strcpy(exp_text,"SysFmtCurrTime(\"%s\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);
#endif

/*unique*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"unique");
   strcpy(exp_text,"unique a");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   varsup = expeval_varsup_open(context,"a",resolve,NULL,varvalue,&va);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_BOOLEAN)
      lprintf("The expression %s is expected to be boolean.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

   for (i = 0; i < 10; i++)
```

```
      {
      va = i;
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("[%s] = [%s], a = %d\n",
            exp_text,value_to_string(context,&exp->value),va);
      }

   for (i = 5; i < 15; i++)
      {
      va = i;
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("[%s] = [%s], a = %d\n",
            exp_text,value_to_string(context,&exp->value),va);
      }

   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*div*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"unless with div");
   strcpy(exp_text,"-1 unless (12345.57 div (a/2))");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   varsup = expeval_varsup_open(context,"a",resolve,NULL,varvalue,&va);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_BOOLEAN)
      lprintf("The expression %s is expected to be numeric.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

   for (i = 0; i < 10; i++)
      {
      va = i;
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("[%s] = [%s], a = %d\n",
            exp_text,value_to_string(context,&exp->value),va);
      }

   for (i = 5; i < 15; i++)
```

```
      {
      va = i;
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("[%s] = [%s], a = %d\n",
            exp_text,value_to_string(context,&exp->value),va);
      }

   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysFmtTime()  %s is not defined on MVS */
#ifndef __HOS_MVS__
   lprintf("Start of test %3.3d: %s\n",++tstnum,
         "SysNumber with SysFmtCurrTime");
   strcpy(exp_text,"SysNumber(SysFmtCurrTime('%s')) mod 86400");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   varsup = expeval_varsup_open(context,"a",resolve,NULL,varvalue,&va);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be numeric.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));

   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);
#endif

/* SysFmtCurrTime between 08h00 and 20h00 */
#ifndef __HOS_MVS__
   lprintf("Start of test %3.3d: %s\n",++tstnum,
         "SysNumber with SysFmtCurrTime 2");
   strcpy(exp_text,
         "((SysNumber(SysFmtCurrTime('%s')) mod 86400) > 28800) and "
         "((SysNumber(SysFmtCurrTime('%s')) mod 86400) < 72000)");
```

```
#else
   strcpy(exp_text,
         "((SysNumber(SysFmtCurrTime('%H%M'))) > 800) and "
         "((SysNumber(SysFmtCurrTime('%H%M'))) < 2000)");
#endif
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   varsup = expeval_varsup_open(context,"a",resolve,NULL,varvalue,&va);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_BOOLEAN)
      lprintf("The expression %s is expected to be boolean.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));

   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/* unless conditional expression - RHS works */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"unless");
   strcpy(exp_text,"1 unless 2");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be numeric.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
```

```
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));

   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/* unless conditional expression - RHS fails */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"unless 2");
   strcpy(exp_text,"-1 unless (1/0)");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_NUMBER)
      lprintf("The expression %s is expected to be numeric.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));

   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/* ifelse true */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"ifelse");
   strcpy(exp_text,"ifelse(1=1,\"Yes this is true\",\"Problem\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
```

```
   if (rc) error_exit(context,FROM(__LINE__));

   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));

   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/* ifelse false */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"ifelse 2");
   strcpy(exp_text,"ifelse(1=0,\"Problem\",\"Not true is OK\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));

   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysStrPadLeft()*/
   /* Padlength > current length */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrPadLeft");
   strcpy(exp_text,"SysStrPadLeft(\"1234567890\",15,\"#\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
```

```
rc = expeval_type_expression(context,exp);
if (rc) error_exit(context,FROM(__LINE__));
if (exp->value.type != EVAL_VALUE_STRING)
   lprintf("The expression %s is expected to be string.\n",exp_text);
rc = expeval_prime(context,exp);
if (rc) error_exit(context,FROM(__LINE__));
rc = expeval_evaluate(context,exp);
if (rc) error_exit(context,FROM(__LINE__));
lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
rc = expeval_final(context,exp);
if (rc) error_exit(context,FROM(__LINE__));
expeval_cleanup(exp);
expeval_context_close(context);

/* Padlength = current length */
lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrPadLeft 2");
strcpy(exp_text,"SysStrPadLeft(\"1234567890\",10,\"#\")");
lprintf("Checking [%s];\n",exp_text);
context = expeval_context_open(0,NULL,NULL);
exp = expeval_compile(context,exp_text);
if (exp) lprintf("Compile should return a node --- good!\n");
else
   {
   lprintf("Invalid compile did not return a node --- bad!\n");
   lprintf("Last error: %s\n",context->last_error);
   }
rc = expeval_type_expression(context,exp);
if (rc) error_exit(context,FROM(__LINE__));
if (exp->value.type != EVAL_VALUE_STRING)
   lprintf("The expression %s is expected to be string.\n",exp_text);
rc = expeval_prime(context,exp);
if (rc) error_exit(context,FROM(__LINE__));
rc = expeval_evaluate(context,exp);
if (rc) error_exit(context,FROM(__LINE__));
lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
rc = expeval_final(context,exp);
if (rc) error_exit(context,FROM(__LINE__));
expeval_cleanup(exp);
expeval_context_close(context);

/* Padlength < current length */
lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrPadLeft 3");
strcpy(exp_text,"SysStrPadLeft(\"1234567890\",5,\"#\")");
lprintf("Checking [%s];\n",exp_text);
context = expeval_context_open(0,NULL,NULL);
exp = expeval_compile(context,exp_text);
if (exp) lprintf("Compile should return a node --- good!\n");
else
   {
   lprintf("Invalid compile did not return a node --- bad!\n");
   lprintf("Last error: %s\n",context->last_error);
   }
```

```
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysStrPadRight()*/
   /* Padlength > current length */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrPadRight");
   strcpy(exp_text,"SysStrPadRight(\"1234567890\",15,\"#\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

   /* Padlength = current length */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrPadRight 2");
   strcpy(exp_text,"SysStrPadRight(\"1234567890\",10,\"#\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
```

```
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

   /* Padlength < current length */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrPadRight 3");
   strcpy(exp_text,"SysStrPadRight(\"1234567890\",5,\"#\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysStrCondPack() - replacement spaces */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrCondPack");
   strcpy(exp_text,"SysStrCondPack('X\"4141410000\"',\" \")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
```

```
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysStrTrimRight()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrTrimRight");
   strcpy(exp_text,"SysStrTrimRight(\"1234567890#####\",\"#\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   tmp = value_to_string(context,&exp->value);
   lprintf("[%s] = [%d][%s]\n",exp_text,strlen(tmp),tmp);
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*strtrimright()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"strtrimright - no trim");
   strcpy(exp_text,"strtrimright(\"1234567890\",\"#\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
```

```
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   tmp = value_to_string(context,&exp->value);
   lprintf("[%s] = [%d][%s]\n",exp_text,strlen(tmp),tmp);
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*rtrim()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"rtrim only trim");
   strcpy(exp_text,"rtrim(\"#####\",\"#\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   tmp = value_to_string(context,&exp->value);
   lprintf("[%s] = [%d][%s]\n",exp_text,strlen(tmp),tmp);
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysStrTrimRight()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrTrimRight - empty string");
   strcpy(exp_text,"SysStrTrimRight(\"\",\"#\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
```

```
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   tmp = value_to_string(context,&exp->value);
   lprintf("[%s] = [%d][%s]\n",exp_text,strlen(tmp),tmp);
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysStrTrimLeft()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrTrimLeft");
   strcpy(exp_text,"SysStrTrimLeft(\"#####1234567890\",\"#\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   tmp = value_to_string(context,&exp->value);
   lprintf("[%s] = [%d][%s]\n",exp_text,strlen(tmp),tmp);
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*strtrimleft()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"strtrimleft - no trim");
   strcpy(exp_text,"strtrimleft(\"1234567890\",\"#\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
```

```
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   tmp = value_to_string(context,&exp->value);
   lprintf("[%s] = [%d][%s]\n",exp_text,strlen(tmp),tmp);
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*ltrim()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"ltrim - only trim");
   strcpy(exp_text,"ltrim(\"#####\",\"#\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   tmp = value_to_string(context,&exp->value);
   lprintf("[%s] = [%d][%s]\n",exp_text,strlen(tmp),tmp);
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysStrTrimLeft()*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrTrimLeft - empty string");
   strcpy(exp_text,"SysStrTrimLeft(\"\",\"#\")");
   lprintf("Checking [%s];\n",exp_text);
```

```
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   tmp = value_to_string(context,&exp->value);
   lprintf("[%s] = [%d][%s]\n",exp_text,strlen(tmp),tmp);
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*SysStrCondPack() - replacement deafult = "?" */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"SysStrCondPack 2");
   strcpy(exp_text,"SysStrCondPack('X\"4141410000\"',\"\")");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*uuid_time_gen() - libuuid time based  uuid generation */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"uuid_time_gen()");
   strcpy(exp_text,"uuid_time_gen()");
```

```
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*uuid_time_gen() - libuuid time based  uuid generation
 * Using with em-dash U+2014 */
   lprintf("Start of test %3.3d: %s\n",++tstnum,
         "strcat('xyz',gsub('-','_',uuid_time_gen(),'g'))");
   strcpy(exp_text,"strcat('xyz',gsub('-','_',uuid_time_gen(),'g'))");
   lprintf("Checking [%s];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string.\n",exp_text);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
```

```
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("[%s] = [%s]\n",exp_text,value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*sfget() 1 */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"sfget");
   strcpy(exp_text,"sfget(\"219Postilion::MetaData275211FWSerialNbr111\
15SWRel11119CommsType11118TermType11115OSVer11116SWHash111211FWSerialNbr\
221010001000000001002242315SWRel2131401E2014060219CommsType\
214INTERNAL MODEM18TermType18EFTsmart15OSVer1982003607816SWHash\
18B4E1963A\",\"FWSerialNbr\")");
   lprintf("Checking [%s] = [1];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string. It is %s\n",
            exp_text,type_names[exp->value.type]);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("======>[%s]\n",value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*sfget() 2 - Look for first item */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"sfget 2");
   strcpy(exp_text,"sfget(\"219Postilion::MetaData275211FWSerialNbr111\
15SWRel11119CommsType11118TermType11115OSVer11116SWHash111211FWSerialNbr\
```

```
221010001000000001002242315SWRel2131401E2014060219CommsType\
214INTERNAL MODEM18TermType18EFTsmart15OSVer1982003607816SWHash\
18B4E1963A\",\"Postilion::MetaData\")");
   lprintf("Checking [%s] = [1];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string. It is %s\n",
            exp_text,type_names[exp->value.type]);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("======>[%s]\n",value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*sfget() 3 - Look for last item */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"sfget 3");
   strcpy(exp_text,"sfget(\"219Postilion::MetaData275211FWSerialNbr111\
15SWRel11119CommsType11118TermType11115OSVer11116SWHash111211FWSerialNbr\
221010001000000001002242315SWRel2131401E2014060219CommsType\
214INTERNAL MODEM18TermType18EFTsmart15OSVer1982003607816SWHash\
18B4E1963A\",\"SWHash\")");
   lprintf("Checking [%s] = [1];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string. It is %s\n",
            exp_text,type_names[exp->value.type]);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
```

```
   lprintf("======>[%s]\n",value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*sfget() 4 - Look for invalid name */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"sfget 4");
   strcpy(exp_text,"sfget(\"219Postilion::MetaData275211FWSerialNbr111\
15SWRel11119CommsType11118TermType11115OSVer11116SWHash111211FWSerialNbr\
2210100010000000100224231 5SWRel2131401E2014060219CommsType\
214INTERNAL MODEM18TermType18EFTsmart15OSVer1982003607816SWHash\
18B4E1963A\",\"XYZ123\")");
   lprintf("Checking [%s] = [1];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string. It is %s\n",
            exp_text,type_names[exp->value.type]);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("======>[%s]\n",value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*sfget() 5 - Look for name where CommsType has invalid length*/
   lprintf("Start of test %3.3d: %s\n",++tstnum,"sfget 5");
   strcpy(exp_text,"sfget(\"219Postilion::MetaData275211FWSerialNbr111\
15SWRel11119CommsType11118TermType11115OSVer11116SWHash111211FWSerialNbr\
2210100010000000100224231 5SWRel2131401E2014060219CommsType\
215INTERNAL MODEM18TermType18EFTsmart15OSVer1982003607816SWHash\
18B4E1963A\",\"XYZ123\")");
   lprintf("Checking [%s] = [1];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
```

```
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string. It is %s\n",
            exp_text,type_names[exp->value.type]);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc)
      lprintf("Return Code %d, correct for negative test\n",rc);
   lprintf("======>[%s]\n",value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*sfset() 1 - Update FWSerialNbr with longer value. */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"sfget 6");
   strcpy(exp_text,"sfset(\"219Postilion::MetaData275211FWSerialNbr111\
15SWRel11119CommsType11118TermType11115OSVer11116SWHash111211FWSerialNbr\
2210100010000000100224231 5SWRel2131401E2014060219CommsType\
214INTERNAL MODEM18TermType18EFTsmart15OSVer1982003607816SWHash\
18B4E1963A\",\"FWSerialNbr\",\"<----------LongerValue---------->\")");
   lprintf("Checking [%s] = [1];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string. It is %s\n",
            exp_text,type_names[exp->value.type]);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("======>[%s]\n",value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*sfset() 2 - Update FWSerialNbr with shorter value. */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"sfget 7");
   strcpy(exp_text,"sfset(\"219Postilion::MetaData275211FWSerialNbr111\
15SWRel11119CommsType11118TermType11115OSVer11116SWHash111211FWSerialNbr\
```

```
2210100010000000001002242315SWRel2131401E2014060219CommsType\
214INTERNAL MODEM18TermType18EFTsmart15OSVer1982003607816SWHash\
18B4E1963A\",\"FWSerialNbr\",\"<--SV-->\")");
   lprintf("Checking [%s] = [1];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string. It is %s\n",
            exp_text,type_names[exp->value.type]);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("======>[%s]\n",value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*sfset() 3 - Update field that does not occur ExampleField with value. */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"sfget 8");
   strcpy(exp_text,"sfset(\"219Postilion::MetaData275211FWSerialNbr111\
15SWRel11119CommsType11118TermType11115OSVer11116SWHash111211FWSerialNbr\
2210100010000000001002242315SWRel2131401E2014060219CommsType\
214INTERNAL MODEM18TermType18EFTsmart15OSVer1982003607816SWHash\
18B4E1963A\",\"ExampleField\",\"<--V-->\")");
   lprintf("Checking [%s] = [1];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string. It is %s\n",
            exp_text,type_names[exp->value.type]);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
```

```
   lprintf("======>[%s]\n",value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*sfset() 4 - Update the first field with a value longer than 100 bytes. */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"sfget 9");
   strcpy(exp_text,"sfset(\"219Postilion::MetaData275211FWSerialNbr111\
15SWRel11119CommsType11118TermType11115OSVer11116SWHash111211FWSerialNbr\
221010001000000001002242315SWRel2131401E2014060219CommsType\
214INTERNAL MODEM18TermType18EFTsmart15OSVer1982003607816SWHash\
18B4E1963A\",\"Postilion::MetaData\",\"<--------><--------><--------><-------->\
<--------><--------><--------><--------><--------><--------><-------->\")");
   lprintf("Checking [%s] = [1];\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string. It is %s\n",
            exp_text,type_names[exp->value.type]);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("======>[%s]\n",value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*gsub() 1 - gsub(r,s,t,h) replace regex r with string s in text t using
 * method h
 */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"gsub");
   strcpy(exp_text,
         "gsub(\"\\(.\\+\\) \\(.\\+\\)\",\"\\2 \\1\",\"ABC DEF\",\"1\")");
   lprintf("Checking [%s]\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
```

```
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string. It is %s\n",
            exp_text,type_names[exp->value.type]);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("======>[%s]\n",value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*gsub() 2 - gsub(r,s,t,h) replace regex r with string s in text t using
 * method h
 * Only replace 3rd occurrence
 */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"gsub 2");
   strcpy(exp_text,
         "gsub(\"\\(ab\\+\\)\",\"xyz\",\"1ab  2abbb 3ab 4abbb\",\"3\")");
   lprintf("Checking [%s]\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string. It is %s\n",
            exp_text,type_names[exp->value.type]);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("======>[%s]\n",value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*gsub() 3 - gsub(r,s,t,h) replace regex r with string s in text t using
 * method h
 * Replace ALL occurrence
 */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"gsub 3");
```

```
   strcpy(exp_text,
         "gsub(\"\\(ab\\+\\)\",\"xyz\",\"1ab  2abbb 3ab 4abbb\",\"g\")");
   lprintf("Checking [%s]\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string. It is %s\n",
            exp_text,type_names[exp->value.type]);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("======>[%s]\n",value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*gsub() 4 - gsub(r,s,t,h) replace regex r with string s in text t using
 * method h
 * Replace ALL with nothing
 */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"replace");
   lprintf("%s\n","==========================================================>");
   strcpy(exp_text,
         "replace(\".*\",\"\",\"abcdefghijklmnopqrst\",\"g\")");
   lprintf("Checking [%s]\n",exp_text);
   context = expeval_context_open(0,NULL,NULL);
   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      }
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string. It is %s\n",
            exp_text,type_names[exp->value.type]);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   rc = expeval_evaluate(context,exp);
   if (rc)
```

```
      lprintf("Return Code %d, correct for negative test\n",rc);
   lprintf("======>[%s]\n",value_to_string(context,&exp->value));
   rc = expeval_final(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   expeval_cleanup(exp);
   expeval_context_close(context);

/*gsub() 5 - gsub(r,s,t,h) replace regex r with string s in text t using
 * method h
 * Test with varsupplier of variables
 */
   lprintf("%s\n","==========================================================>");
   lprintf("Start of test %3.3d: %s\n",++tstnum,"gsub variable");
  /* Create a context in which all expeval library calls can be made.
   */
   context = expeval_context_open(0,NULL,NULL);

  /* Define variable suppliers. One for each of the variable variables
   * in the expression.
   */
   strncpy(gsub_varname[0],"regex",sizeof(gsub_varname[0]));
   strncpy(gsub_varname[1],"replacement",sizeof(gsub_varname[1]));
   strncpy(gsub_varname[2],"text",sizeof(gsub_varname[2]));
   strncpy(gsub_varname[3],"how",sizeof(gsub_varname[3]));
   gsub_varname[4][0] = 0;

   strncpy(gsub_varvalue[0],"\\([^ ]\\+\\) \\([^ ]\\+\\)",
         sizeof(gsub_varvalue[0]));
   strncpy(gsub_varvalue[1],"\\2 \\1",sizeof(gsub_varvalue[1]));
   strncpy(gsub_varvalue[2],"ABC DEF",sizeof(gsub_varvalue[2]));
   strncpy(gsub_varvalue[3],"g",sizeof(gsub_varvalue[3]));
   gsub_varvalue[4][0] = 0;

   rc = 0;
   for (i=0; gsub_varname[i][0]; i++)
      {
      lprintf("expeval_varsup_open %s => %s\n",gsub_varname[i],
            gsub_varvalue[i]);
      varsup = expeval_varsup_open(context,gsub_varname[i],resolve_string,NULL,
            varstring,&gsub_varvalue[i]);
      if (!varsup) rc++;
      }
   if (rc) error_exit(context,FROM(__LINE__));

   strcpy(exp_text,"replace(regex,replacement,text,how)");
   lprintf("Checking [%s]\n",exp_text);

   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
```

```
      lprintf("Last error: %s\n",context->last_error);
      error_exit(context,FROM(__LINE__));
      }
   if (exp)
      {
      rc = expeval_type_expression(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      if (exp->value.type != EVAL_VALUE_STRING)
         lprintf("The expression %s is expected to be string. It is %s\n",
               exp_text,type_names[exp->value.type]);
      rc = expeval_prime(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));

     /* Evaluate the statement based on the current values.
      */
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("[%s]\t\t======>[%s]\n",gsub_varvalue[2],
            value_to_string(context,&exp->value));

     /* Do a second one.
      */
      strncpy(gsub_varvalue[2],"UVW XYZ",sizeof(gsub_varvalue[2]));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("[%s]\t\t======>[%s]\n",gsub_varvalue[2],
            value_to_string(context,&exp->value));

     /* and another
      */
      strncpy(gsub_varvalue[2],"1234567 XYZ",sizeof(gsub_varvalue[2]));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("[%s]\t\t======>[%s]\n",gsub_varvalue[2],
            value_to_string(context,&exp->value));

     /* and another
      */
      strncpy(gsub_varvalue[2],"UVWXYZ 1234567890",sizeof(gsub_varvalue[2]));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("[%s]\t======>[%s]\n",gsub_varvalue[2],
            value_to_string(context,&exp->value));

     /* and another swap all occurrences
      */
      strncpy(gsub_varvalue[2],"A1 bA1 A2 BA2",sizeof(gsub_varvalue[2]));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("[%s]\t\t======>[%s]\n",gsub_varvalue[2],
            value_to_string(context,&exp->value));
```

```
     /* and another swap all occurrences
      */
      strncpy(gsub_varvalue[2],"A1 bA1 A2 BA2",sizeof(gsub_varvalue[2]));
      strncpy(gsub_varvalue[3],"2",sizeof(gsub_varvalue[3]));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("[%s]\t\t======>[%s]\n",gsub_varvalue[2],
            value_to_string(context,&exp->value));

     /* Finalise and cleanup the statement.
      */
      rc = expeval_final(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      expeval_cleanup(exp);
      }
   expeval_context_close(context);

/*alias() 1 - alias(table,search) return looked up value in table using search.
 * Test with varsupplier of variables
 */
   lprintf("%s\n","==========================================================>");
   lprintf("Start of test %3.3d: %s\n",++tstnum,"alias");
  /* Create a context in which all expeval library calls can be made.
   */
   context = expeval_context_open(0,NULL,NULL);

  /* Define variable suppliers. One for each of the variable variables
   * in the expression.
   */
   strncpy(alias_varname,"search",sizeof(alias_varname));
   strncpy(alias_varvalue,"F",sizeof(alias_varvalue));

   rc = 0;
   lprintf("expeval_varsup_open %s => value @%p\n",alias_varname,
         alias_varvalue);
   varsup = expeval_varsup_open(context,alias_varname,resolve_string,NULL,
         varstring,&alias_varvalue);
   if (!varsup) error_exit(context,FROM(__LINE__));

   strcpy(exp_text,"alias(\"F=Female,M=Male,O=Other\",search)");
   lprintf("Checking [%s]\n",exp_text);

   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      error_exit(context,FROM(__LINE__));
      }
   if (exp)
      {
```

```
      rc = expeval_type_expression(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      if (exp->value.type != EVAL_VALUE_STRING)
         lprintf("The expression %s is expected to be string. It is %s\n",
               exp_text,type_names[exp->value.type]);
      rc = expeval_prime(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));

     /* Evaluate the statement based on the current values.
      */
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

     /* Do a second one.
      */
      strncpy(alias_varvalue,"O",sizeof(alias_varvalue));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

     /* and another
      */
      strncpy(alias_varvalue,"M",sizeof(alias_varvalue));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

     /* and another
      */
      strncpy(alias_varvalue,"X",sizeof(alias_varvalue));
      rc = expeval_evaluate(context,exp);
      if (rc)
         lprintf("Return Code %d, correct for negative test\n",rc);
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

     /* and another
      */
      strncpy(alias_varvalue,"F",sizeof(alias_varvalue));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

      expeval_cleanup(exp);
      }
   expeval_context_close(context);

/*alias() 2 - alias(table,search) return looked up value in table using search.
 * Test with varsupplier of variables and lookup file
 */
   lprintf("%s\n","=========================================================>");
   lprintf("Start of test %3.3d: %s\n",++tstnum,"alias 2");
```

```
      {
      FILE *lookup;

      lookup = fopen("lookup.txt","wb");
      fputs("F=Female\n",lookup);
      fputs("O=Other\n",lookup);
      fputs("M=Male\n",lookup);
      fputs("X=CrossOver\n",lookup);
      fputs("U=Unknown\n",lookup);
      fputs("A=Androgonous\n",lookup);
      fclose(lookup);
      }

   /* Create a context in which all expeval library calls can be made.
    */
    context = expeval_context_open(0,NULL,NULL);

   /* Define variable suppliers. One for each of the variable variables
    * in the expression.
    */
    strncpy(alias_varname,"search",sizeof(alias_varname));
    strncpy(alias_varvalue,"F",sizeof(alias_varvalue));

    rc = 0;
    lprintf("expeval_varsup_open %s => value @%p\n",alias_varname,
          alias_varvalue);
    varsup = expeval_varsup_open(context,alias_varname,resolve_string,NULL,
          varstring,&alias_varvalue);
    if (!varsup) error_exit(context,FROM(__LINE__));

    strcpy(exp_text,"lookup(\"./lookup.txt\",search)");
    lprintf("Checking [%s]\n",exp_text);

    exp = expeval_compile(context,exp_text);
    if (exp) lprintf("Compile should return a node --- good!\n");
    else
       {
       lprintf("Invalid compile did not return a node --- bad!\n");
       lprintf("Last error: %s\n",context->last_error);
       error_exit(context,FROM(__LINE__));
       }
    if (exp)
       {
       rc = expeval_type_expression(context,exp);
       if (rc) error_exit(context,FROM(__LINE__));
       if (exp->value.type != EVAL_VALUE_STRING)
          lprintf("The expression %s is expected to be string. It is %s\n",
                exp_text,type_names[exp->value.type]);
       rc = expeval_prime(context,exp);
       if (rc) error_exit(context,FROM(__LINE__));
```

```
     /* Evaluate the statement based on the current values.
      */
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

     /* Do a second one.
      */
      strncpy(alias_varvalue,"O",sizeof(alias_varvalue));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

     /* and another
      */
      strncpy(alias_varvalue,"M",sizeof(alias_varvalue));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

     /* and another
      */
      strncpy(alias_varvalue,"Z",sizeof(alias_varvalue));
      rc = expeval_evaluate(context,exp);
      if (rc)
         lprintf("Return Code %d, correct for negative test\n",rc);

      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

     /* and another
      */
      strncpy(alias_varvalue,"X",sizeof(alias_varvalue));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

     /* and another
      */
      strncpy(alias_varvalue,"F",sizeof(alias_varvalue));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

      expeval_cleanup(exp);
      }
   expeval_context_close(context);

/*alias() 3 - alias(table,search) return looked up value in table using search.
 * Test with varsupplier of variables and lookup file
 */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"alias 3");
   lprintf("%s\n","=========================================================>");
```

```
      {
      FILE *lookup;

      lookup = fopen("lookup.txt","wb");
      fputs("001=First\n",lookup);
      fputs("099=Almost100\n",lookup);
      fputs("010=ten\n",lookup);
      fputs("101=CenturyPlus\n",lookup);
      fputs("1001=Dalmations\n",lookup);
      fputs("10001=Large\n",lookup);
      fclose(lookup);
      }

   /* Create a context in which all expeval library calls can be made.
    */
    context = expeval_context_open(0,NULL,NULL);

   /* Define variable suppliers. One for each of the variable variables
    * in the expression.
    */
    strncpy(alias_varname,"search_this",sizeof(alias_varname));
    strncpy(alias_varvalue,"101",sizeof(alias_varvalue));

    rc = 0;
    lprintf("expeval_varsup_open %s => value @%p\n",alias_varname,
          alias_varvalue);
    varsup = expeval_varsup_open(context,alias_varname,resolve_string,NULL,
          varstring,&alias_varvalue);
    if (!varsup) error_exit(context,FROM(__LINE__));

    strcpy(exp_text,"alias(\"./lookup.txt\",search_this)");
    lprintf("Checking [%s]\n",exp_text);

    exp = expeval_compile(context,exp_text);
    if (exp) lprintf("Compile should return a node --- good!\n");
    else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      error_exit(context,FROM(__LINE__));
      }
    if (exp)
      {
      rc = expeval_type_expression(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      if (exp->value.type != EVAL_VALUE_STRING)
         lprintf("The expression %s is expected to be string. It is %s\n",
               exp_text,type_names[exp->value.type]);
      rc = expeval_prime(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
```

```
     /* Evaluate the statement based on the current values.
      */
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

     /* Do a second one.
      */
      strncpy(alias_varvalue,"1001",sizeof(alias_varvalue));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

     /* and another
      */
      strncpy(alias_varvalue,"099",sizeof(alias_varvalue));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

     /* and another
      */
      strncpy(alias_varvalue,"010",sizeof(alias_varvalue));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

     /* and another
      */
      strncpy(alias_varvalue,"2",sizeof(alias_varvalue));
      rc = expeval_evaluate(context,exp);
      if (rc)
         lprintf("Return Code %d, correct for negative test\n",rc);
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

     /* and another
      */
      strncpy(alias_varvalue,"10001",sizeof(alias_varvalue));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

      expeval_cleanup(exp);
      }
   expeval_context_close(context);

/*alias() 4 - alias(table,search) return looked up value in table using search.
 * Test with varsupplier of variables and lookup file and three lookups.
 */
   lprintf("Start of test %3.3d: %s\n",++tstnum,"alias 4");
   lprintf("%s\n","=========================================================>");
```

```
    {
    FILE *lookup;

    lookup = fopen("lookup.txt","wb");
    fputs("001=First\n",lookup);
    fputs("099=Almost100\n",lookup);
    fputs("010=ten\n",lookup);
    fputs("101=CenturyPlus\n",lookup);
    fputs("1001=Dalmations\n",lookup);
    fputs("10001=Large\n",lookup);
    fclose(lookup);
    }

/* Create a context in which all expeval library calls can be made.
 */
 context = expeval_context_open(0,NULL,NULL);

/* Define variable suppliers. One for each of the variable variables
 * in the expression.
 */
 strncpy(alias_varname,"search_this",sizeof(alias_varname));
 strncpy(alias_varname2,"search_this2",sizeof(alias_varname2));
 strncpy(alias_varname3,"search_this3",sizeof(alias_varname3));
 strncpy(alias_varvalue,"101",sizeof(alias_varvalue));
 strncpy(alias_varvalue2,"10001",sizeof(alias_varvalue2));
 strncpy(alias_varvalue3,"099",sizeof(alias_varvalue3));

 rc = 0;
 lprintf("expeval_varsup_open %s => value @%p\n",alias_varname,
       alias_varvalue);
 varsup = expeval_varsup_open(context,alias_varname,resolve_string,NULL,
       varstring,&alias_varvalue);
 if (!varsup) error_exit(context,FROM(__LINE__));
 lprintf("expeval_varsup_open %s => value @%p\n",alias_varname2,
       alias_varvalue2);
 varsup = expeval_varsup_open(context,alias_varname2,resolve_string,NULL,
       varstring,&alias_varvalue2);
 if (!varsup) error_exit(context,FROM(__LINE__));
 lprintf("expeval_varsup_open %s => value @%p\n",alias_varname3,
       alias_varvalue3);
 varsup = expeval_varsup_open(context,alias_varname3,resolve_string,NULL,
       varstring,&alias_varvalue3);
 if (!varsup) error_exit(context,FROM(__LINE__));

 strcpy(exp_text,"strcat(alias(\"./lookup.txt\",search_this),"
       "strcat(alias(\"./lookup.txt\",search_this2),"
       "alias(\"./lookup.txt\",search_this3)))");
 lprintf("Checking [%s]\n",exp_text);

 exp = expeval_compile(context,exp_text);
 if (exp) lprintf("Compile should return a node --- good!\n");
 else
```

```
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      error_exit(context,FROM(__LINE__));
      }
   if (exp)
      {
      rc = expeval_type_expression(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      if (exp->value.type != EVAL_VALUE_STRING)
         lprintf("The expression %s is expected to be string. It is %s\n",
               exp_text,type_names[exp->value.type]);
      rc = expeval_prime(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));

     /* Evaluate the statement based on the current values.
      */
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s:%s:%s=>[%s]\n",alias_varvalue,alias_varvalue2,alias_varvalue3,
            value_to_string(context,&exp->value));

     /* Do a second one.
      */
      strncpy(alias_varvalue,"1001",sizeof(alias_varvalue));
      strncpy(alias_varvalue2,"101",sizeof(alias_varvalue2));
      strncpy(alias_varvalue3,"001",sizeof(alias_varvalue3));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s:%s:%s=>[%s]\n",alias_varvalue,alias_varvalue2,alias_varvalue3,
            value_to_string(context,&exp->value));

      expeval_cleanup(exp);
      }
   expeval_context_close(context);

/*isalias() 1 - alias(table,search) return looked up value in table using
 * search.  Test with varsupplier of variables
 */
   lprintf("%s\n","==========================================================>");
   lprintf("Start of test %3.3d: %s\n",++tstnum,"isalias 4");
  /* Create a context in which all expeval library calls can be made.
   */
   context = expeval_context_open(0,NULL,NULL);

  /* Define variable suppliers. One for each of the variable variables
   * in the expression.
   */
   strncpy(alias_varname,"search",sizeof(alias_varname));
   strncpy(alias_varvalue,"F",sizeof(alias_varvalue));

   rc = 0;
```

```
lprintf("expeval_varsup_open %s => value @%p\n",alias_varname,
      alias_varvalue);
varsup = expeval_varsup_open(context,alias_varname,resolve_string,NULL,
      varstring,&alias_varvalue);
if (!varsup) error_exit(context,FROM(__LINE__));

strcpy(exp_text,"isalias(\"F=Female,M=Male,O=Other\",search)");
lprintf("Checking [%s]\n",exp_text);

exp = expeval_compile(context,exp_text);
if (exp) lprintf("Compile should return a node --- good!\n");
else
   {
   lprintf("Invalid compile did not return a node --- bad!\n");
   lprintf("Last error: %s\n",context->last_error);
   error_exit(context,FROM(__LINE__));
   }
if (exp)
   {
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_BOOLEAN)
      lprintf("The expression %s is expected to be boolean. "
            "It is %s\n",exp_text,type_names[exp->value.type]);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

  /* Evaluate the statement based on the current values.
   */
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

  /* Do a second one.
   */
   strncpy(alias_varvalue,"O",sizeof(alias_varvalue));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

  /* and another
   */
   strncpy(alias_varvalue,"M",sizeof(alias_varvalue));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

  /* and another
   */
   strncpy(alias_varvalue,"X",sizeof(alias_varvalue));
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
```

```
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

     /* and another
      */
      strncpy(alias_varvalue,"F",sizeof(alias_varvalue));
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",alias_varvalue,value_to_string(context,&exp->value));

      expeval_cleanup(exp);
      }
   expeval_context_close(context);

/*pstore_set and pstore_get() 1 - pstore_set(host,variable,num_value)
 * Set a persistent store numeric value and retrieve it again.
 */
   lprintf("%s\n","=========================================================>");
   if (skip_pstore) goto skip_pstore_tests;
   lprintf("Start of test %3.3d: %s\n",++tstnum,"pstore");
   lprintf("%s\n","Is PSTORE running on the localhost:60061 [n|y]?");
   rc = getchar();
   if ((char)rc != 'y') goto skip_pstore_tests;

  /* Create a context in which all expeval library calls can be made.
   */
   context = expeval_context_open(0,NULL,NULL);

  /* Define variable suppliers. One for each of the variables in the expression.
   */
   strncpy(pstore_varname,"keyword",sizeof(pstore_varname));
   strncpy(pstore_varvalue,"myvalue",sizeof(pstore_varvalue));
   strncpy(pstore_hostname,"host",sizeof(pstore_hostname));
   strncpy(pstore_hostvalue,"localhost:60061",sizeof(pstore_hostvalue));
   strncpy(pstore_svalname,"value",sizeof(pstore_svalname));
   strncpy(pstore_svalvalue,"17",sizeof(pstore_svalvalue));

   rc = 0;
   lprintf("sup %s=>@%p:%s\n",pstore_varname,pstore_varvalue,pstore_varvalue);
   varsup = expeval_varsup_open(context,pstore_varname,resolve_string,NULL,
         varstring,&pstore_varvalue);
   if (!varsup) error_exit(context,FROM(__LINE__));
   lprintf("sup %s=>@%p:%s\n",pstore_hostname,pstore_hostvalue,
         pstore_hostvalue);
   varsup = expeval_varsup_open(context,pstore_hostname,resolve_string,NULL,
         varstring,&pstore_hostvalue);
   if (!varsup) error_exit(context,FROM(__LINE__));
   lprintf("sup %s=>@%p:%s\n",pstore_svalname,&pstore_varvalue,
         pstore_svalvalue);
   varsup = expeval_varsup_open(context,pstore_svalname,resolve_string,NULL,
         varstring,&pstore_svalvalue);
   if (!varsup) error_exit(context,FROM(__LINE__));
```

```
strcpy(exp_text,"pstore_set(host,keyword,value)");
lprintf("Checking [%s]\n",exp_text);

exp = expeval_compile(context,exp_text);
if (exp) lprintf("Compile should return a node --- good!\n");
else
   {
   lprintf("Invalid compile did not return a node --- bad!\n");
   lprintf("Last error: %s\n",context->last_error);
   error_exit(context,FROM(__LINE__));
   }
if (exp)
   {
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be STRING. "
            "It is %s\n",exp_text,type_names[exp->value.type]);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

  /* Evaluate the statement based on the current values.
   */
   lprintf("Port %s should fail\n",pstore_hostvalue);
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("%s=>[%s]\n",pstore_varvalue,
         value_to_string(context,&exp->value));

   strncpy(pstore_hostvalue,"localhost:60060",sizeof(pstore_hostvalue));
   lprintf("\nPort %s should work\n",pstore_hostvalue);
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("%s=>[%s]\n",pstore_varvalue,
         value_to_string(context,&exp->value));

   expeval_cleanup(exp);
   }
expeval_context_close(context);

/* Now retrieve that same variable and check that it is the same.
 * BUT do use a new context.
 */
lprintf("%s\n","==========================================================>");
lprintf("Start of test %3.3d: %s\n",++tstnum,"pstore 2");
/* Create a context in which all expeval library calls can be made.
 */
context = expeval_context_open(0,NULL,NULL);

/* Define variable suppliers. One for each of the variables in the expression.
 */
strncpy(pstore_varname,"keyword",sizeof(pstore_varname));
```

```
 strncpy(pstore_varvalue,"myvalue",sizeof(pstore_varvalue));
 strncpy(pstore_hostname,"host",sizeof(pstore_hostname));
/* Set the host and port. Leave of the port to test the default value 60060.
 */
 strncpy(pstore_hostvalue,"localhost",sizeof(pstore_hostvalue));

 rc = 0;
 lprintf("sup %s=>@%p:%s\n",pstore_varname,pstore_varvalue,pstore_varvalue);
 varsup = expeval_varsup_open(context,pstore_varname,resolve_string,NULL,
       varstring,&pstore_varvalue);
 if (!varsup) error_exit(context,FROM(__LINE__));
 lprintf("sup %s=>@%p:%s\n",pstore_hostname,pstore_hostvalue,
       pstore_hostvalue);
 varsup = expeval_varsup_open(context,pstore_hostname,resolve_string,NULL,
       varstring,&pstore_hostvalue);
 if (!varsup) error_exit(context,FROM(__LINE__));

 strcpy(exp_text,"pstore_get(host,keyword)");
 lprintf("Checking [%s]\n",exp_text);

 exp = expeval_compile(context,exp_text);
 if (exp) lprintf("Compile should return a node --- good!\n");
 else
    {
    lprintf("Invalid compile did not return a node --- bad!\n");
    lprintf("Last error: %s\n",context->last_error);
    error_exit(context,FROM(__LINE__));
    }
 if (exp)
    {
    rc = expeval_type_expression(context,exp);
    if (rc) error_exit(context,FROM(__LINE__));
    if (exp->value.type != EVAL_VALUE_STRING)
       lprintf("The expression %s is expected to be STRING. "
             "It is %s\n",exp_text,type_names[exp->value.type]);
    rc = expeval_prime(context,exp);
    if (rc) error_exit(context,FROM(__LINE__));

   /* Evaluate the statement based on the current values.
    */
    rc = expeval_evaluate(context,exp);
    if (rc) error_exit(context,FROM(__LINE__));
    lprintf("%s=>[%s]\n",pstore_varvalue,
          value_to_string(context,&exp->value));
    lprintf("pstore_set(\"myvalue\",\"17\") %s pstore_get(\"myvalue\")\n",
          (strcmp(pstore_svalvalue,value_to_string(context,&exp->value))) ?
          "!=" : "==");

    expeval_cleanup(exp);
    }
 expeval_context_close(context);
```

```
/*pstore_set and pstore_get() 2 -
 * pstore_set(host,variable,num_value) unless pstore_get(host,variable)
 * Set a persistent store numeric value and retrieve it again.
 */
   lprintf("%s\n","=========================================================>");
   lprintf("Start of test %3.3d: %s\n",++tstnum,"pstore 3");
  /* Create a context in which all expeval library calls can be made.
   */
   context = expeval_context_open(0,NULL,NULL);

  /* Define variable suppliers. One for each of the variables in the expression.
   */
   strncpy(pstore_varname,"keyword",sizeof(pstore_varname));
   strncpy(pstore_varvalue,"AKeywordThatNeverExists",sizeof(pstore_varvalue));
   strncpy(pstore_hostname,"host",sizeof(pstore_hostname));
   strncpy(pstore_hostvalue,"localhost:60060",sizeof(pstore_hostvalue));
   strncpy(pstore_svalname,"value",sizeof(pstore_svalname));
   strncpy(pstore_svalvalue,"17",sizeof(pstore_svalvalue));

   rc = 0;
   lprintf("sup %s=>@%p:%s\n",pstore_varname,pstore_varvalue,pstore_varvalue);
   varsup = expeval_varsup_open(context,pstore_varname,resolve_string,NULL,
         varstring,&pstore_varvalue);
   if (!varsup) error_exit(context,FROM(__LINE__));
   lprintf("sup %s=>@%p:%s\n",pstore_hostname,
         pstore_hostvalue,pstore_hostvalue);
   varsup = expeval_varsup_open(context,pstore_hostname,resolve_string,NULL,
         varstring,&pstore_hostvalue);
   if (!varsup) error_exit(context,FROM(__LINE__));
   lprintf("sup %s=>@%p:%s\n",pstore_svalname,&pstore_varvalue,
         pstore_svalvalue);
   varsup = expeval_varsup_open(context,pstore_svalname,resolve_string,NULL,
         varstring,&pstore_svalvalue);
   if (!varsup) error_exit(context,FROM(__LINE__));

   strcpy(exp_text,
         "pstore_set(host,keyword,value) unless pstore_get(host,keyword)");
   lprintf("Checking [%s]\n",exp_text);

   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      error_exit(context,FROM(__LINE__));
      }
   if (exp)
      {
      rc = expeval_type_expression(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      if (exp->value.type != EVAL_VALUE_STRING)
```

```
        lprintf("The expression %s is expected to be STRING. "
              "It is %s\n",exp_text,type_names[exp->value.type]);
      rc = expeval_prime(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));

     /* Evaluate the statement based on the current values.
      */
      lprintf("To reset:\ncmlpstore -h localhost -n AKeywordThatNeverExists "
            "-u\n\n");
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",pstore_varvalue,
            value_to_string(context,&exp->value));

     /*
      * do not use expeval_cleanup as it causes a double free error (perhaps
      * because variables are used twice in two different functions). I think
      * that perhaps expeval_cleanup should not be used when everything works as
      * expeval_context_close will clean up.
      */
      }
   expeval_context_close(context);

/* pstore_get() 3 - * pstore_get_cset(host,variable,num_value)
 * Set a persistent store numeric value and retrieve it again.
 */
   lprintf("%s\n","==========================================================>");
   lprintf("Start of test %3.3d: %s\n",++tstnum,"pstore 4");
  /* Create a context in which all expeval library calls can be made.
   */
   context = expeval_context_open(0,NULL,NULL);

  /* Define variable suppliers. One for each of the variables in the expression.
   */
   strncpy(pstore_varname,"keyword",sizeof(pstore_varname));
   strncpy(pstore_varvalue,"keyword3",sizeof(pstore_varvalue));
   strncpy(pstore_hostname,"host",sizeof(pstore_hostname));
   strncpy(pstore_hostvalue,"localhost:60060",sizeof(pstore_hostvalue));
   strncpy(pstore_svalname,"value",sizeof(pstore_svalname));
   strncpy(pstore_svalvalue,"17",sizeof(pstore_svalvalue));

   rc = 0;
   lprintf("sup %s=>@%p:%s\n",pstore_varname,pstore_varvalue,pstore_varvalue);
   varsup = expeval_varsup_open(context,pstore_varname,resolve_string,NULL,
         varstring,&pstore_varvalue);
   if (!varsup) error_exit(context,FROM(__LINE__));
   lprintf("sup %s=>@%p:%s\n",pstore_hostname,pstore_hostvalue,
         pstore_hostvalue);
   varsup = expeval_varsup_open(context,pstore_hostname,resolve_string,NULL,
         varstring,&pstore_hostvalue);
   if (!varsup) error_exit(context,FROM(__LINE__));
   lprintf("sup %s=>@%p:%s\n",pstore_svalname,&pstore_varvalue,
```

```
         pstore_svalvalue);
   varsup = expeval_varsup_open(context,pstore_svalname,resolve_string,NULL,
         varstring,&pstore_svalvalue);
   if (!varsup) error_exit(context,FROM(__LINE__));

   strcpy(exp_text,"pstore_get_cset(host,keyword,value)");
   lprintf("Checking [%s]\n",exp_text);

   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
      lprintf("Last error: %s\n",context->last_error);
      error_exit(context,FROM(__LINE__));
      }
   if (exp)
      {
      rc = expeval_type_expression(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      if (exp->value.type != EVAL_VALUE_STRING)
         lprintf("The expression %s is expected to be STRING. "
               "It is %s\n",exp_text,type_names[exp->value.type]);
      rc = expeval_prime(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));

     /* Evaluate the statement based on the current values.
      */
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",pstore_varvalue,
            value_to_string(context,&exp->value));

      expeval_cleanup(exp);
      }
   expeval_context_close(context);

/* pstore_get() 4 - pstore_get_incr(host,variable)
 * Get a persistent store numeric value and retrieve its increment again.
 */
   lprintf("%s\n","==========================================================>");
   lprintf("Start of test %3.3d: %s\n",++tstnum,"pstore 5");
  /* Create a context in which all expeval library calls can be made.
   */
   context = expeval_context_open(0,NULL,NULL);

  /* Define variable suppliers. One for each of the variables in the expression.
   */
   strncpy(pstore_varname,"keyword",sizeof(pstore_varname));
   strncpy(pstore_varvalue,"keyword3",sizeof(pstore_varvalue));
   strncpy(pstore_hostname,"host",sizeof(pstore_hostname));
   strncpy(pstore_hostvalue,"localhost:60060",sizeof(pstore_hostvalue));
```

```
 rc = 0;
 lprintf("sup %s=>@%p:%s\n",pstore_varname,pstore_varvalue,pstore_varvalue);
 varsup = expeval_varsup_open(context,pstore_varname,resolve_string,NULL,
       varstring,&pstore_varvalue);
 if (!varsup) error_exit(context,FROM(__LINE__));
 lprintf("sup %s=>@%p:%s\n",pstore_hostname,pstore_hostvalue,
       pstore_hostvalue);
 varsup = expeval_varsup_open(context,pstore_hostname,resolve_string,NULL,
       varstring,&pstore_hostvalue);
 if (!varsup) error_exit(context,FROM(__LINE__));

/* Use unless to set the first value.
 */
 strcpy(exp_text,"pstore_get_incr(host,keyword)");
 lprintf("Checking [%s]\n",exp_text);

 exp = expeval_compile(context,exp_text);
 if (exp) lprintf("Compile should return a node --- good!\n");
 else
    {
    lprintf("Invalid compile did not return a node --- bad!\n");
    lprintf("Last error: %s\n",context->last_error);
    error_exit(context,FROM(__LINE__));
    }
 if (exp)
    {
    rc = expeval_type_expression(context,exp);
    if (rc) error_exit(context,FROM(__LINE__));
    if (exp->value.type != EVAL_VALUE_NUMBER)
       lprintf("The expression %s is expected to be number. "
             "It is %s\n",exp_text,type_names[exp->value.type]);
    rc = expeval_prime(context,exp);
    if (rc) error_exit(context,FROM(__LINE__));

   /* Evaluate the statement based on the current values.
    */
    lprintf("To reset:\ncmlpstore -h localhost -n keyword3 -V17\n\n");
    lprintf("Retrieve first value\n");
    rc = expeval_evaluate(context,exp);
    if (rc) error_exit(context,FROM(__LINE__));
    lprintf("%s=>[%s]\n",pstore_varvalue,
          value_to_string(context,&exp->value));

    lprintf("Retrieve second value\n");
    rc = expeval_evaluate(context,exp);
    if (rc) error_exit(context,FROM(__LINE__));
    lprintf("%s=>[%s]\n",pstore_varvalue,
          value_to_string(context,&exp->value));

    lprintf("Retrieve third value\n");
    rc = expeval_evaluate(context,exp);
```

```
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",pstore_varvalue,
            value_to_string(context,&exp->value));

      expeval_cleanup(exp);
      }
   expeval_context_close(context);

/* pstore_get() 5 - pstore_get_incr_cset(host,variable,value)
 * Get a persistent store numeric value and retrieve its increment again.
 */
   lprintf("%s\n","=========================================================>");
   lprintf("Start of test %3.3d: %s\n",++tstnum,"pstore 6");
  /* Create a context in which all expeval library calls can be made.
   */
   context = expeval_context_open(0,NULL,NULL);

  /* Define variable suppliers. One for each of the variables in the expression.
   */
   strncpy(pstore_varname,"keyword",sizeof(pstore_varname));
   strncpy(pstore_varvalue,"keyword5",sizeof(pstore_varvalue));
   strncpy(pstore_hostname,"host",sizeof(pstore_hostname));
   strncpy(pstore_hostvalue,"localhost:60060",sizeof(pstore_hostvalue));
   strncpy(pstore_nvalname,"value",sizeof(pstore_nvalname));
   pstore_nvalvalue=17;

   rc = 0;
   lprintf("sup %s=>@%p:%s\n",pstore_varname,pstore_varvalue,pstore_varvalue);
   varsup = expeval_varsup_open(context,pstore_varname,resolve_string,NULL,
         varstring,&pstore_varvalue);
   if (!varsup) error_exit(context,FROM(__LINE__));
   lprintf("sup %s=>@%p:%s\n",pstore_hostname,pstore_hostvalue,
         pstore_hostvalue);
   varsup = expeval_varsup_open(context,pstore_hostname,resolve_string,NULL,
         varstring,&pstore_hostvalue);
   if (!varsup) error_exit(context,FROM(__LINE__));
   lprintf("sup %s=>@%p:%d\n",pstore_nvalname,&pstore_varvalue,
         pstore_nvalvalue);
   varsup = expeval_varsup_open(context,pstore_nvalname,resolve,NULL,
         varvalue,&pstore_nvalvalue);
   if (!varsup) error_exit(context,FROM(__LINE__));

  /* Use unless to set the first value.
   */
   strcpy(exp_text,"pstore_get_incr_cset(host,keyword,value)");
   lprintf("Checking [%s]\n",exp_text);

   exp = expeval_compile(context,exp_text);
   if (exp) lprintf("Compile should return a node --- good!\n");
   else
      {
      lprintf("Invalid compile did not return a node --- bad!\n");
```

```
      lprintf("Last error: %s\n",context->last_error);
      error_exit(context,FROM(__LINE__));
      }
   if (exp)
      {
      rc = expeval_type_expression(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      if (exp->value.type != EVAL_VALUE_NUMBER)
         lprintf("The expression %s is expected to be number. "
               "It is %s\n",exp_text,type_names[exp->value.type]);
      rc = expeval_prime(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));

     /* Evaluate the statement based on the current values.
      */
      lprintf("To reset:\ncmlpstore -h localhost -n keyword5 -u\n\n");
      lprintf("Retrieve first value\n");
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",pstore_varvalue,
            value_to_string(context,&exp->value));

      lprintf("Retrieve second value\n");
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",pstore_varvalue,
            value_to_string(context,&exp->value));

      lprintf("Retrieve third value\n");
      rc = expeval_evaluate(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      lprintf("%s=>[%s]\n",pstore_varvalue,
            value_to_string(context,&exp->value));

      expeval_cleanup(exp);
      }
   expeval_context_close(context);
   goto end_pstore;

skip_pstore_tests:
   lprintf("%s\n","Skipped pstore tests due to response or commandline");
end_pstore:

/* Value Call back for variables in an expression. This sets up the call back
 * and will print the expressions variable's names and values in the callback
 * Test with varsupplier of variables
 */
   lprintf("%s\n","=========================================================>");
   lprintf("Start of test %3.3d: %s\n",++tstnum,"variable value callback");
  /* Create a context in which all expeval library calls can be made.
   */
   context = expeval_context_open(0,NULL,NULL);
```

```
/* Define variable suppliers. One for each of the variable variables
 * in the expression.
 */
 strncpy(VC_varname[0],"VarS1",sizeof(VC_varname[0]));
 strncpy(VC_varname[1],"VarS2",sizeof(VC_varname[1]));
 strncpy(VC_varname[2],"VarS3",sizeof(VC_varname[2]));
 strncpy(VC_varname[3],"VarS4",sizeof(VC_varname[3]));
 strncpy(VC_varname[4],"VarS5",sizeof(VC_varname[4]));
 strncpy(VC_varname[5],"VarI1",sizeof(VC_varname[5]));
 strncpy(VC_varname[6],"VarI2",sizeof(VC_varname[6]));
 strncpy(VC_varname[7],"VarI3",sizeof(VC_varname[7]));
 strncpy(VC_varname[8],"VarI4",sizeof(VC_varname[8]));
 strncpy(VC_varname[9],"VarI5",sizeof(VC_varname[9]));

 strncpy(VC_varvalueS[0],"ValueS 1",sizeof(VC_varvalueS[0]));
 strncpy(VC_varvalueS[1],"ValueS 2",sizeof(VC_varvalueS[1]));
 strncpy(VC_varvalueS[2],"ValueS 3",sizeof(VC_varvalueS[2]));
 strncpy(VC_varvalueS[3],"ValueS 4",sizeof(VC_varvalueS[3]));
 strncpy(VC_varvalueS[4],"ValueS 5",sizeof(VC_varvalueS[4]));
 VC_varvalueI[0] = 0;                    /* Will be set later */
 VC_varvalueI[1] = 2000;
 VC_varvalueI[2] = 30000;
 VC_varvalueI[3] = 400000;
 VC_varvalueI[4] = 5000000;

 rc = 0;
 for (i=0; i<5; i++)
    {
    lprintf("expeval_varsup_open %s => %s\n",VC_varname[i],
          VC_varvalueS[i]);
    varsup = expeval_varsup_open(context,VC_varname[i],resolve_string,NULL,
          varstring,&VC_varvalueS[i]);
    if (!varsup) rc++;
    }
 if (rc) error_exit(context,FROM(__LINE__));

 rc = 0;
 for (i=5, j=0; i<10; i++, j++)
    {
    lprintf("expeval_varsup_open %s => %d\n",VC_varname[i],
          VC_varvalueI[j]);
    varsup = expeval_varsup_open(context,VC_varname[i],resolve,NULL,
          varvalue,&VC_varvalueI[j]);
    if (!varsup) rc++;
    }
 if (rc) error_exit(context,FROM(__LINE__));

 strcpy(exp_text,"strcat(VarS1,strcat(VarS2,strcat(VarS3,strcat(VarS4,"
       "strcat(VarS5,strcat(' :',string(VarI1+VarI2+VarI3+VarI4+"
       "VarI5)))))))");
 lprintf("Checking [%s]\n",exp_text);
```

```
exp = expeval_compile(context,exp_text);
if (exp) lprintf("Compile should return a node --- good!\n");
else
   {
   lprintf("Invalid compile did not return a node --- bad!\n");
   lprintf("Last error: %s\n",context->last_error);
   error_exit(context,FROM(__LINE__));
   }
if (exp)
   {
   rc = expeval_type_expression(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   if (exp->value.type != EVAL_VALUE_STRING)
      lprintf("The expression %s is expected to be string. It is %s\n",
            exp_text,type_names[exp->value.type]);
   rc = expeval_prime(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));

  /* Evaluate 3 times the statement based on the current values and updating
   * varvalue[0] 3 times.
   * 1st time - no velue call back set
   * 2nd time - value call backe set - output of each variable should be seen
   * 3rd time - no value call back set - as it is first unregistered.
   */
   lprintf("%s\n","====================== About to call (with no registered "
         "value callback) expeval_evaluate()");
   VC_varvalueI[0] = 100;
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("Result=[%s]\n",value_to_string(context,&exp->value));

   lprintf("====================== %s\n","Call expeval_evaluate() complete");

   lprintf("====================== %s\n","About to call expeval_evaluate()");
   lprintf("====================== %s\n",
         "Regestering a value callback routine");
   VC_varvalueI[0] = 200;
   expeval_register_value_callback(context,VC_user_data,VC_callback);
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
   lprintf("Result=[%s]\n",value_to_string(context,&exp->value));

   lprintf("====================== %s\n","Call expeval_evaluate() complete");

   lprintf("====================== %s\n",
         "About to call (with un-registered value callback) "
         "expeval_evaluate()");
   expeval_unregister_value_callback(context);
   VC_varvalueI[0] = 300;
   rc = expeval_evaluate(context,exp);
   if (rc) error_exit(context,FROM(__LINE__));
```

```
      lprintf("Result=[%s]\n",value_to_string(context,&exp->value));

      lprintf("===================== %s\n","Call expeval_evaluate() complete");

     /* Finalise and cleanup the statement.
      */
      rc = expeval_final(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      expeval_cleanup(exp);
      }
   expeval_context_close(context);

/******************************************************************************
******************************************************************************
******************************************************************************/

/*big expression*/
   if (big_expression)
      {
      big_text = malloc(50000000);
      big_text[0] = 0;
      strcpy(big_text,"(a = -1)");
      for (i = 0; i < 1000000; i++)
         {
         sprintf(exp_text,"or (a = %d)",i);
         strcat(big_text+strlen(big_text),exp_text);
         }

      lprintf("Start of test %3.3d: %s\n",++tstnum,"BIG EXPRESSION");
      lprintf("Checking [%s];\n",big_text);
      context = expeval_context_open(0,NULL,NULL);
      varsup = expeval_varsup_open(context,"a",resolve,NULL,varvalue,&va);
      exp = expeval_compile(context,big_text);
      if (exp) lprintf("Compile should return a node --- good!\n");
      else
         {
         lprintf("Invalid compile did not return a node --- bad!\n");
         lprintf("Last error: %s\n",context->last_error);
         }
      rc = expeval_type_expression(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      if (exp->value.type != EVAL_VALUE_BOOLEAN)
         lprintf("The expression %s is expected to be boolean.\n",
               big_text);
      rc = expeval_prime(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));

      for (i = 0; i < 10; i++)
         {
         va = i;
         rc = expeval_evaluate(context,exp);
         if (rc) error_exit(context,FROM(__LINE__));
```

```
         lprintf("[%s] = [%s], a = %d\n",
               big_text,value_to_string(context,&exp->value),va);
         }

      for (i = 5; i < 15; i++)
         {
         va = i;
         rc = expeval_evaluate(context,exp);
         if (rc) error_exit(context,FROM(__LINE__));
         lprintf("[%s] = [%s], a = %d\n",
               big_text,value_to_string(context,&exp->value),va);
         }

      rc = expeval_final(context,exp);
      if (rc) error_exit(context,FROM(__LINE__));
      expeval_cleanup(exp);
      expeval_context_close(context);
      }
   else
      lprintf("\n*****\n%s\n*****\n",
            "Big expression test bypassed. Use --big-expression, -b to run");

   exit(0);
   } /* main */

  /* Function error_exit() prints the error message in the supplied context
   * and exits the program.
   */

static void error_exit(expeval_context_t *context, char *message)
   {
   char s[10];

   s[0] = 0;
   lprintf(__FILE__ " Error| %s\n\n",context->last_error);
   if (message) lprintf("%s\n",message);
   if (prompt)
      {
      lprintf("%s:","What should happen? (<enter>=Continue, a=Abort)");
      fgets(s,sizeof(s),stdin);
      if (s[0] == 'a') exit(16);
      }
   else
      if (!force) exit(16);
   } /* error_exit */

static char *value_to_string(expeval_context_t *context,
      expeval_value_t *value)
   {
   static char buffer[32760];
   int rc;
```

```
   rc = expeval_value_to_string(context,value,sizeof(buffer),buffer);
   if (rc < 0)
      sprintf(buffer,"Error: %s.",context->last_error);

   return buffer;
   } /* value_to_string */

static int resolve(expeval_context_t *context, expeval_variable_t *variable)
   {
  /* Copy the supplied variable address (int *) from the variable
   * supplier to the variable instance.
   */
   variable->varvalue_private_data = variable->resolve_private_data;

  /* Set the type of the variable to a number:
   */
   variable->value.type = EVAL_VALUE_NUMBER;

   return 0;
   } /* resolve */

static int varvalue(expeval_context_t *context, expeval_variable_t *variable)
   {
   int *varval;            /* address of int supplying variable value */
   char string[20];        /* string formatted version of variable value */
   int rc;

   varval = variable->varvalue_private_data;
   sprintf(string,"%d",*varval);
   rc = expeval_varsup_insert_number(context,variable,string);
   if (rc) error_exit(context,"Error in varvalue");

   return 0;
   } /* varvalue */

static int resolve_string(expeval_context_t *context,
      expeval_variable_t *variable)
   {
  /* Copy the supplied variable address (char *) from the variable
   * supplier to the variable instance.
   */
   variable->varvalue_private_data = variable->resolve_private_data;

  /* Set the type of the variable to a number:
   */
   variable->value.type = EVAL_VALUE_STRING;

   return 0;
   } /* resolve_string */

static int varstring(expeval_context_t *context, expeval_variable_t *variable)
   {
```

```
   char *varval;           /* address of string supplying variable value */
   int rc;

   varval = variable->varvalue_private_data;
   rc = expeval_varsup_insert_storage(context,variable,strlen(varval),varval);
   if (rc) error_exit(context,"Error in varstring");

   return 0;
   } /* varstriing */

static int funentry(expeval_context_t *context, expeval_function_t *function,
            expeval_value_t *value, expeval_node_t *arguments)
   {
   char string[32760];      /* string format of function arguments */
   int argint[3];           /* integer format of function arguments */
   expeval_node_t *arg;     /* argument to the function */
   int i;
   int rc;

  /* extract the evaluated arguments of the function:
   */

   for (i = 0, arg = arguments; arg; arg = arg->peer, i++)
      {
      rc = expeval_value_to_int(context,&arg->value,&argint[i]);
      if (rc < 0) return -1;
      }

   sprintf(string,"%d",argint[0]+argint[1]+argint[2]);

   rc = expeval_value_number(context,string,value);
   if (rc) error_exit(context,"Error in funentry");

   return 0;
   } /* funentry */

int list_fields(objtype_t *objtype, objfield_t *objfield, void *callback_parm)
   {
   lprintf("Field = %s\n",objfield->name);
   return 0;
   } /* list_fields */

static int lprintf(char *format, ...)
   {
   va_list ap;
   char text[32760];

   va_start(ap,format);
   vsnprintf(text,sizeof(text),format,ap);
   va_end(ap);
   printf("TEST%3.3d: %s",tstnum,text);
   } /* lprintf */
```

```
  /* Format the incoming line number nicely and return the string.
   */
static char *FROM(int l)
   {
   static char s[1024];
   snprintf(s,sizeof(s),"Last error at program (%s) line %d",__FILE__,l);
   return s;
   }

  /* Function VC_callback() is called by expeval each time it handles a variable
   * in and expression under evaluation if and only if the routine has been
   * registered as a value callback with expeval_register_value_callback(). This
   * routine is called once for each variable and receives the name and value of
   * the variable as strings irrespective of the internal expeval type of the
   * variable.
   */

static void VC_callback(void *user_data, char *keyword, char *value)
   {
   fprintf(stderr,"\t%s","In Value Callback: ");
   fprintf(stderr,"Keyword=%s, Value=%s\n",keyword,value);

   return;
   } /* VC_callback */
```

Running this program gives the result:

```
[stephen@nomad expeval]$ ./example
Answer to expression evaluation when a = 15 and b = 25.
1.3+f(a,b,4)*10 = 441.3
Correct answer  = 441.300000
Answer to expression evaluation when a = 15 and b = 25.
1.3+f(a,b,4)*10 = 441.3
Correct answer  = 441.300000
Answer to expression evaluation when a = 150 and b = 250.
1.3+f(a,b,4)*10 = 4041.3
Correct answer  = 4041.300000
```

7 Expression Grammar

The string expression passed to the `expeval_compile()` function to compile into an expression tree must conform to a specific grammar. This grammar is described in this section with the assistance of rail diagrams. In addition, the semantics of the expressions is partly determine by operator precedence and associativity. In most cases, this precedence and associativity confirms to that of commonly encountered in algebra, but the full precedence is included for completeness.

The interpretation of the expression in the supplied string is governed by the grammar, operator precedence and associativity to build an expression tree corresponding to that specific expression string.

Any environment variable expansion is applied before the contents of the supplied string is analysed for its lexical elements. Environment variable references must conform to the pattern `${[_A-Za-z][_A-Za-z0-9]*}`.

7.1 Lexical Elements

The elements of an expression are the variables, literals, operators and other character symbols used to from an expression. These lexical elements or *tokens* are often single characters having their own apparent meaning, but some are grouped together to form a word having a specific meaning. Included or associated each each token may be an *attribute value*. In the above, the reference to characters having their own apparent meaning can be stated as the attribute value of these individual characters is the character itself. Other tokens are formed by sequences of characters conforming to a *pattern* and the string of characters matching such a pattern is referred to as a *lexeme*. For example, a pattern may determine that sequence of characters is a string literal token, in this case the lexeme is the full sequence of characters including any surrounding quotation marks present to indicate the string literal, and the attribute value is the sequence of characters inside the quotation marks after processing any escape characters present [1].

The above comments are general, but specific to `expeval` the following tokens are processed as part of expression strings as indicated. We choose to use regular expressions to describe the patterns to which these lexemes must conform.

- *Number*. A *Number* is a literal whose attribute value is suitable for use in arithmetic operations. The lexemes for *Number* tokens must match the regular expression `{(\+|-)?[0-9]+(\.[0-9]+)?` with the attribute value being the corresponding number. In other words, numbers start with an optional sign (with unsigned numbers assumed to be positive), followed by a sequence of one or more digits, these in turn may optionally be followed by a decimal point and a further sequence of one or more digits.

  Examples of a *Number* are: 0, 1, 3.141 10000.0 and 0.1234, but not .1234 or 10000..

- *String*. A *String* is a literal whose attribute value is suitable for string and relational operators. The lexemes for *String* tokens must match one of the regular expressions `\"[\^\"\n]+\"` or `{\'[\^\'\n]+\'`. A *String* may also be indicated as a sequence of paired characters representing the hexadecimal digits. The lexeme for such a *String* must match one of the regular expressions
  `(x|X)\'([0-9a-fA-F][0-9a-fA-F])+\'` or
  `(x|X)\"([0-9a-fA-F][0-9a-fA-F])+\"`.

In other words a string literal can be specified as a sequence of characters enclosed in either quotation marks or apostrophes. Strings may not contain their delimiting apostrophe or quotation marks and may not contain the new-line character. A *String* indicated using enclosing quotation marks or apostrophes has an attribute value determined by removing the enclosing characters. Additionally, a string literal may be specified as sequence of hexadecimal paired digits. In this case the characters within the quotation marks are required to be paired and restricted to the hexadecimal digits `0` to `9` and `A` to `F` (or `a` to `f`). Such a hexadecimal literal string is distinguished from the first form of string literal by a preceding `X` or `x` in front of the opening quotation mark or apostrophe.

Examples of a *String* are
`"Hello World!","It's time",'Hello World!',X'01020304'` or `x"01020304"`.

- *Identifier*. An *Identifier* is token whose lexeme must conform to one of the patterns `[A-Za-z][_A-Za-z0-9]*` or `[0-9]+[_A-Za-z][_A-Za-z0-9]*`. In other words the lexeme for an identifier should start with an upper or lower case alphabetic character followed by a sequence of one or more underscore characters, upper or lower case alphabetic characters or the decimal digits. Alternatively, a lexeme for an *Identifier* may start with a sequence of one or more decimal digits, followed by an underscore character, an upper or lower case alphabetic character, and followed again by a sequence of one or more underscore characters, upper or lower case alphabetic characters or the decimal digits.

  The attribute value of an *Identifier* token is a reference to the item whose name matches the lexeme pattern. The meaning of the item in terms of types and supplied values is determined by the syntactic context within which the *Identifier* is found (for example, whether it is a reference to a variable or a function), and the semantics of the use of the item is defined by the supplier context in effect for the instance of the `expeval` library. For example, a variable supplier or a function definition.

- All white spaces are ignored and do not form tokens. White spaces include sequences of the space character, new-line characters, the tab character and the line-feed character.

- All other single characters form tokens one their own with the character being used as both the lexeme and the attribute value.

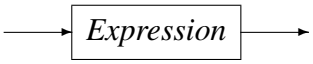
7.2 Syntactic Elements

The required expression syntax is expressed here in terms of rail-road diagrams corresponding to generating production over the lexical elements described above. The description starts with the *start* symbol and refines this recursively into all the terminal

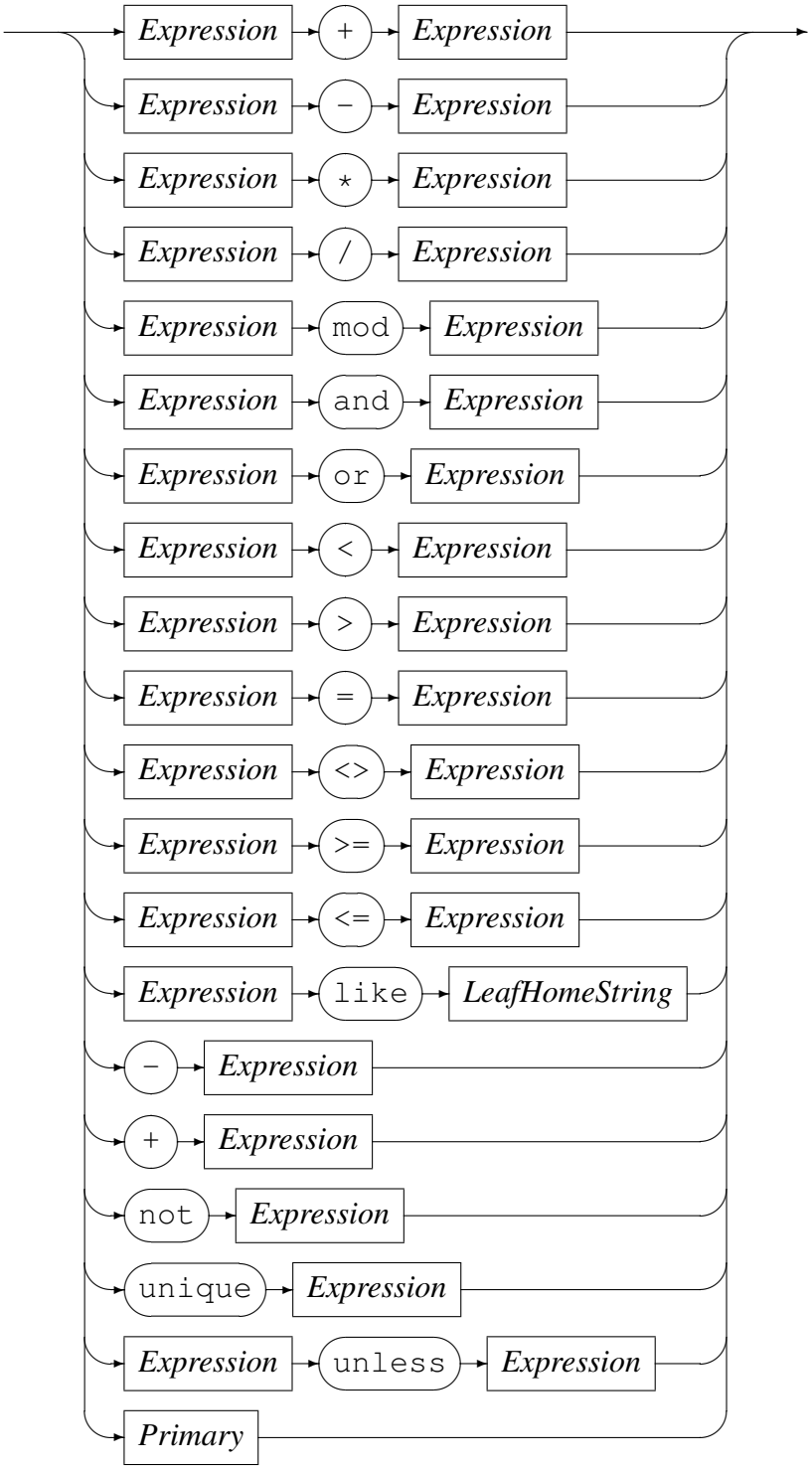
symbol lexical elements and non-terminal syntactical elements. *Terminal symbols* are the lexical tokens which map to the lexemes found in a particular expression string. One or more adjacent terminal symbols may form a syntactical construct grouped together to form a *non-terminal symbol*. Non-terminal symbols may also be constructed of a mixture of both terminal and non-terminal symbols.

The term non-terminal refers to the fact that when using these *production* rules to generate statements in the language by successively replacing symbols (starting with the start symbol), then further replacement of the symbols is possible by applying further production rules. Terminal symbols on the other hand cannot be replaced when generating statements in the language using the production rules.

ExpressionStatement



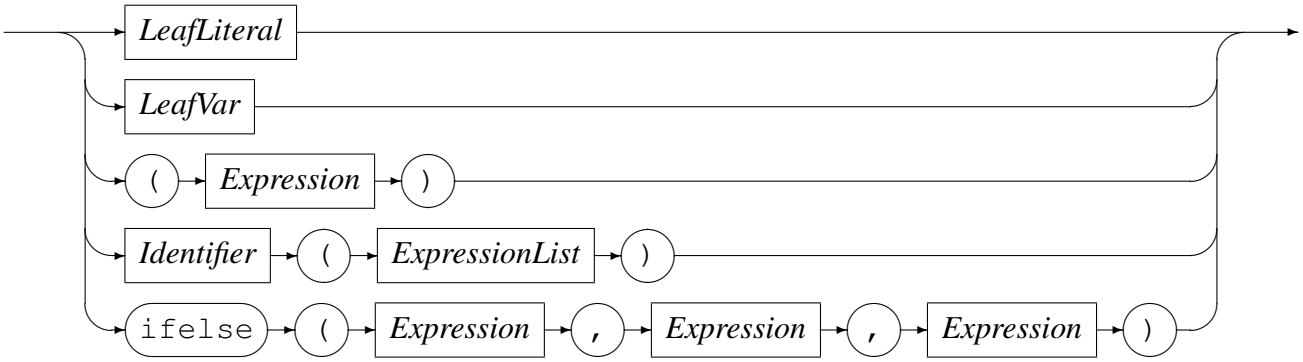
Expression



The `unless`-operator conditionally returns the value of the right-hand operand, unless there is an error evaluating the right-hand operand. In the case where the right-hand operand fails to evaluate to a proper value, the value of the left-hand operand is returned

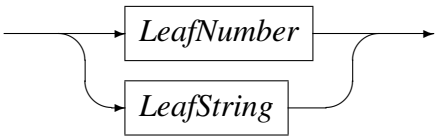
instead. The left-hand operand is always evaluated before the right-hand operand. If the left-hand operand fails to evaluate to a proper value, then the result of the `unless`-operator is a failure.

Primary

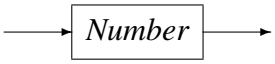


An *Expression* may take the form of a *Primary* symbol. A *Primary* non-terminal symbol in turn may take the form of a *LeafLiteral* symbol, a *LeafVar* symbol, and expression enclosed in parentheses, an *Identifier* followed by an *ExpressionList* in parentheses indicating a function call, or the conditional evaluation operator `ifelse`.

LeafLiteral



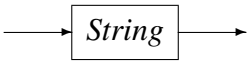
LeafNumber



LeafString

A *LeafLiteral* may be a *String* or a *Number* as described in Section 7.1. Where required by the encoding indicated or defaulted, a characters representing the attribute value of a string are changed to an alternate character set if the required character set is not the same as the home character set being used. For example, on a machine in which the characters are naturally represented using the EBCDIC character set encoding, if the data being processed is from a machine in which the characters are naturally represented using the EBCDIC character set, then the characters in the *String* literal (assumed to be represented in EBCDIC) will be translated to their corresponding ASCII characters for processing. This does not apply to *String* literals that were represented as a sequence of hexadecimal digits.

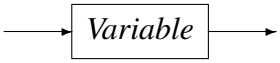
LeafHomeString



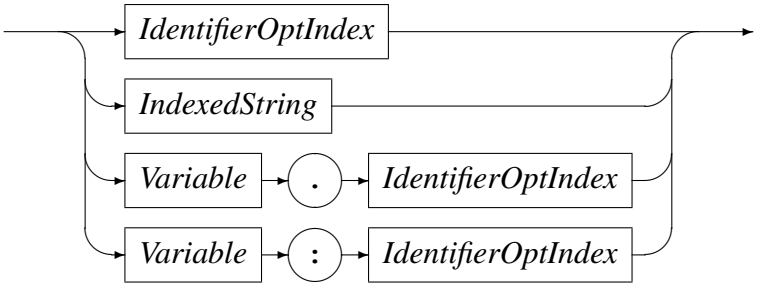
A *LeafHomeString* is a *String* not represented as a sequence of hexadecimal digits, but in which the encoding is left in that of the natural encoding of the machine processing the data — that is the machine on which the expression string is being compiled. This is required for the right-hand operand of the `like` operator as this operator translates the value the left-hand operand into the local encoding in order to check against if there is a pattern match.

An *Expression* may also be formed by two sub-*Expressions* with one of the valid dyadic operators between them, as well as a sub-*Expressions* prefixed by one of the valid monadic operators. See Table 1 for a list of operators, their aridity, precedence and associativity.

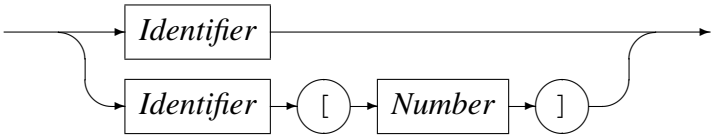
LeafVar



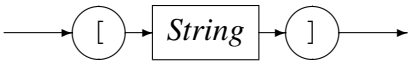
Variable



IdentifierOptIndex

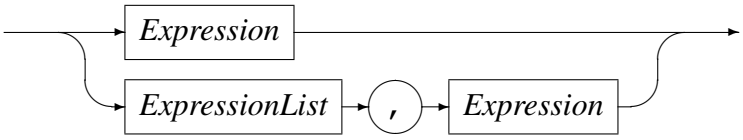


IndexedString



A *LeafVar* is a variable reference which may optionally be qualified and which is followed by an optional index list. The interpretation of the whole of the *LeafVar* as to its validity, its type and to the values that will be required during the evaluation of the expression is left to the variable supplier defined and chosen to deal with the variable.

ExpressionList



An expression list is a comma-separated list of *Expressions* and is the form in which an actual parameter list takes for a function when enclosed in parentheses following a function name *Identifier*.

Table 1 lists the allowed operators, their precedence, associativity, and whether or not they are monadic or dyadic.

Operator	Precedence	Associativity	Aridity	Type
`<>`	1	left	dyadic	Relational
`>=`	1	left	dyadic	Relational
`<=`	1	left	dyadic	Relational
`=`	1	left	dyadic	Relational
`>`	1	left	dyadic	Relational
`<`	1	left	dyadic	Relational
`+`	2	left	dyadic	Arithmetic
`-`	2	left	dyadic	Arithmetic
`or`	2	left	dyadic	Boolean
`*`	3	left	dyadic	Arithmetic
`/`	3	left	dyadic	Arithmetic
`div`	3	left	dyadic	Arithmetic
`and`	3	left	dyadic	Boolean
`mod`	3	left	dyadic	Arithmetic
`-`	4	left	monadic	Arithmetic
`not`	4	left	monadic	Boolean
`unique`	4	left	monadic	boolean
`unless`	4	left	dyadic	Operand

Table 1: Operators: Precedence, Associativity, and Aridity

8 Header File

The interface specification to the `expeval` library is defined in the header file `expeval.h`. This is where any updates and further details of the `expeval` API can be found.

```
#ifndef EXPEVAL_H
#define EXPEVAL_H
  /* File: expeval.h
   *
```

```
 * This is the header file for the expression evaluation library. This
 * is the second version of the expression evaluation library which
 * operates in a more optimal and generic way than does the first
 * version of the expression evaluation library. The efficiency
 * improvement comes from keeping the expression internal node values
 * in an internal form and from keeping the storage for the node values
 * together with the nodes.  The library is made more generic by removing
 * the value extraction method from the library and requiring the user
 * of the library to supply the extraction methods.
 *
 * This file also describes the supplier intervale to the expression
 * evaluation library. Implementations of the supplier interface are
 * required to supply values for the leaf nodes in expressions and are
 * required to supply literal values for compare operations so that the
 * compares can be done optimally.
 *
 * The supplying of literal values is called upon during the preparation
 * phase of expressions and the calling for literal values is done during
 * the execution phase of the expression. In order to bind the method
 * of supplying a value to an leaf node (variable) a call is made during
 * the preparation of the expression. The responsibility of this expression
 * is to supply the method in the node and any private data that this method
 * might require in the node so that the function can be called directly
 * for the required value during expression evaluation.
 *
 * The supplier interface comprises of the types:
 * - expeval_resolve_t
 * - expeval_byteimage_t
 * - expeval_varvalue_t
 * - expeval_supplier_t
 * and the functions:
 * - expeval_varsup_open()
 * - expeval_varsup_close()
 * - expeval_varsup_insert_storage()
 * - expeval_varsup_insert_number()
 * - expeval_varsup_insert_boolean()
 *
 *
 * Author: Stephen Donaldson [stephen@codemagus.com].
 *
 * Copyright (c) 2003 Stephen Donaldson. All Rights Reserved.
 * Copyright (c) 2007--2014 Code Magus Limited. All Rights Reserved.
 */

/*
 * $Author: hayward $
 * $Date: 2022/10/18 15:44:06 $
 * $Id: expeval.h,v 1.29 2022/10/18 15:44:06 hayward Exp $
 * $Name:  $
 * $Revision: 1.29 $
 * $State: Exp $
 *
```

```
* $Log: expeval.h,v $
* Revision 1.29  2022/10/18 15:44:06  hayward
* Allow a caller to de-register the values
* callback function. This allows a caller
* to register a values callback just before
* calling expeval_evaluate() and then
* de-registering it just after. The caller
* can then decide for which evaluate calls
* they need the call back active for or even
* register and de-register different values
* call back routines for different evaluations
* on the same expeval context.
*
* Revision 1.28  2022/09/15 14:18:41  hayward
* Add the ability for a caller to register
* a variable value call back function. This
* then causes expeval to call the function
* for every variable processed in an expression
* within expeval_evaluate(). A user_data pointer
* and a name and value are passed to the routine.
* This allows a user to record the variables and
* their values that contributed to the result of
* an expression.
*
* Revision 1.27  2020/12/20 12:17:03  stephen
* Add power function for integer powers
*
* Revision 1.26  2020/10/31 14:15:29  stephen
* Add random number generator functions to expeval and documentation.
* Three random number generators are added which use numerical recipes
* as the underlying functions. The functions added are runif (for random
* uniformly distributed numbers); rnorm (for normally distributed random
* numbers); and rexp (for exponentially distributed random numbers).
*
* Revision 1.25  2020/10/24 15:59:53  stephen
* Change to use decdigits.h for value of DECNUMDIGITS
*
* Revision 1.24  2017/11/17 17:31:53  hayward
* Add isalias() or inlookup() function that checks
* that a keyword exists in a lookup table and returns
* a boolean true or false.
* Add a hash table into the context structure that
* can hold a pointer to any data area. This allows
* different code paths or instances of the same
* code path to have common private data. The first
* code path to use this is intable(), lookup() and
* inlookup() where the table from argument 1 is
* stored in a hash table and the address of this
* is stored in the hash of hashes keyed on the
* string value of the argument.
*
* Revision 1.23  2017/08/19 10:00:49  stephen
```

```
* Add integer division operator div to expeval
*
* Revision 1.22  2014/12/09 23:58:21  stephen
* Add ifelse-operator and update documentation
*
* Revision 1.21  2014/12/09 20:36:13  stephen
* Update notices in header file
*
* Revision 1.20  2014/09/09 17:16:34  stephen
* Add conditional unless-operator
*
* Revision 1.19  2011/07/31 13:00:44  stephen
* Add support for unique operator
*
* Revision 1.18  2011/01/11 09:19:44  stephen
* Clarify description of function
*
* Revision 1.17  2011/01/11 01:38:26  stephen
* Sort key serialise functions for values
*
* Revision 1.16  2010/09/22 15:03:02  hayward
* Move the callback and callback parameter
* to the context_open function from the
* compile function and make the variables
* part of the context structure. This allows
* both the caller and the expeval library to
* access these values consistently.
*
* Revision 1.15  2010/09/22 13:18:38  hayward
* Correct usage of module statics for semantic errors
* and callback parameter.
*
* Revision 1.14  2010/02/16 15:06:07  stephen
* Allow propagation from supplier structure
*
* Revision 1.13  2010/01/26 12:27:10  stephen
* Update header documentation and expose variable resolve function.
*
* Revision 1.12  2009/11/02 11:35:05  stephen
* Add direct method for evaluating arithmetic negation
*
* Revision 1.11  2009/10/08 20:00:00  hayward
* Convert arg_types to a finite array instead of
* a flexible array using []. This allows it to
* compile on z/OS without warnings.
*
* Revision 1.10  2008/07/29 06:44:11  stephen
* Swap operands around for not/XOR operator.
* Common/shared nodes are not to be freed until context deleted.
*
* Revision 1.9  2008/05/21 09:36:24  stephen
* Cleanup example program expression cleanup
```

```
   *
   * Revision 1.8  2008/04/08 17:58:35  stephen
   * Fixes required comming out of integration into objtypes
   *
   * Revision 1.7  2008/04/08 09:26:04  stephen
   * Add support for object types variable supplier
   *
   * Revision 1.6  2008/04/07 19:44:36  stephen
   * Add support for compiling strings into expression trees
   *
   * Revision 1.5  2008/01/22 11:59:25  stephen
   * Add support support for cleanup of user defined functions and variables
   *
   * Revision 1.4  2008/01/21 09:45:32  stephen
   * Code cleanup and implementation of defined functions
   *
   * Revision 1.3  2008/01/11 09:24:27  stephen
   * Complete initial development of library
   *
   * Revision 1.2  2007/11/25 08:48:40  stephen
   * Changes for conversion to new library
   *
   * Revision 1.1.1.1  2007/11/01 20:56:08  stephen
   * Add Expression Evaluation Library to CVS Repository
   *
   */

#include <hashtab.h>
#include <decdigits.h>

  /*
   * Constants and options:
   */

#ifndef EXPEVAL_NUMBER_TYPE
   #define EXPEVAL_NUMBER_TYPE expeval_internal_number_t
#endif /* EXPEVAL_NUMBER_TYPE */
#define EXPEVAL_DECIMAL_DIGITS DECNUMDIGITS /* internal bytes for decimal number */
#define EXPEVAL_ERROR_MAX  2000 /* max error message string */

  /*
   * Types and datastructures:
   */

typedef struct expeval_context expeval_context_t;
typedef struct value_string value_string_t;
typedef struct value_number value_number_t;
typedef struct expeval_value expeval_value_t;   /* evaluated value type */
typedef struct expeval_variable expeval_variable_t;
typedef struct expeval_node expeval_node_t;    /* expresson node type */
typedef struct expeval_function expeval_function_t;
typedef struct expeval_internal_number expeval_internal_number_t;
```

```
typedef void (*expeval_output_callback_t)(void *callpack_parm, char *text);
typedef void (*expeval_value_callback_t)(void *user_data, char *keyword,
      char *value);

  /*
   * Types for functions required to be implemented by the supplier
   * interface.
   */

  /* The variable resolution function is used to resolve variables and is
   * a function supplied by the user of the expeval library. Both the variable
   * resolution function and the private date are supplied by the user of the
   * library. The variable resolution function is expected to report failures
   * in variable resolution by returning -1 to the caller and is expected
   * to place a meaningful message in last_error. Otherwise the function
   * returns zero. The variable resolution function operates as callback
   * from the expeval library.
   */

typedef int (*expeval_resolve_t)(expeval_context_t *context,
      expeval_variable_t *variable);

  /* The byte image preparation function is used where a literal is compared
   * to a variable. This allows the literal to be formed in a manner which
   * allows the compare operation to be done as a storage compare operation.
   * In order to do this the literal value must be encoded in a manner
   * dictated by the type of the variable.
   */

typedef int (*expeval_byteimage_t)(expeval_context_t *context,
            expeval_node_t *arguments);

  /* The variable value function is called during expression evaluation and
   * is required to insert the current value of the variable into the node
   * of that variable so that the expression evaluation may proceed with the
   * current value of the variable.
   */

typedef int (*expeval_varvalue_t)(expeval_context_t *context,
      expeval_variable_t *variable);

  /* Supplier interface implementations than implement the compare storage
   * function for equality and inequality (those same implementations that
   * supply a byteimage function for this purpose) must also supply a variable
   * address function which will be used during expression evaluation where
   * the compare storage operators need to be called. The function should
   * return the variable address as determined at that point in execution
   * of the expression or should return NULL after placing an appropriate
   * message into the last_error of the supplied context.
   */

typedef void *(*expeval_varaddress_t)(expeval_context_t *context,
```

```
      expeval_variable_t *variable);

  /* A variable structure has an optional variable cleanup method that will
   * be called when the resources for the variable are about to be freed.
   * This function will be called if present and if required the value must
   * be set by a prior call to one of the supplier interface functions
   * provided by the expression evaluation library user implementing the
   * supply of variable values.
   */

typedef void (*expeval_varcleanup_t)(expeval_variable_t *variable);

  /* A function structure has an optional function cleanup method that will
   * be called when the resources for the function are about to be freed.
   * This function will be called if present and if required the value must
   * be set by a prior call or by the registrar of the function.
   */

typedef void (*expeval_funcleanup_t)(expeval_function_t *function);

  /* The function evaluation method is called during evaluation of the
   * expression. The function is called once all the paramaters have been
   * evaluted. The types of the paramaters are checked against the signature
   * of the function during type checking of the containing expression and
   * hence can be trusted to be correct.
   */

typedef int (*expeval_entrypoint_t)(expeval_context_t *context,
      expeval_function_t *function, expeval_value_t *value,
      expeval_node_t *arguments);

  /*
   * Structures and types:
   */

typedef struct expeval_supplier expeval_supplier_t;

struct expeval_supplier
   {
   char *name;                 /* Name of the supplier interface:
                                * The name is also the top level node
                                * name of any variable that belongs to
                                * the supplier interface. So for example
                                * a variable called A.B would have a
                                * supplier interface with the name of A.
                                */

   expeval_resolve_t resolve;  /* Entry point of variable resolve function:
                                * The variable resolve function is called
                                * for each variable belonging to the supplier
                                * interface during expression preparation.
                                * It is the responsibilty of this function
```

```
                                * to plug a value method address in the
                                * given variable node which will be called
                                * during expression execution.
                                */

   expeval_byteimage_t byteimage;
                               /* Entry point to literval value prepare
                                * function:
                                * This function is called for a literal
                                * value that is used to compare against
                                * a value supplied by a variable. This
                                * gives the compare operator the opportunity
                                * to prepare the storage so that a storage
                                * compare can be as part of the execution
                                * of the compare.
                                */

   expeval_varvalue_t varvalue;/* Entry point to variable value supplier
                                * function:
                                * This function is used to supply variable
                                * values during expression evaluation. This
                                * is the default entry point that will be
                                * used to supply the variable values. If this
                                * entry point is missing then the value must
                                * be supplied by the resolve function when it
                                * is called to resolve the variable.
                                */

   void *resolve_private_data; /* Default private data structure for resolver.
                                * This is the value that will be used as the
                                * default private data and will be plugged
                                * into the variable structure before the
                                * resolver is called (if there is one). If this
                                * value is not supplied and one is required,
                                * then the variable resolver must supply a
                                * value for the variable value function to
                                * use.
                                */

   expeval_context_t *context; /* Value to be inherited by the variable as the
                                * actual context to be used in the evaluation
                                * of the variable.
                                */
   };

  /* The internal number representation is a type which is defined just
   * so that the correct number of bytes are reserved for the internal
   * representation of a number.
   */

struct expeval_internal_number
   {
```

```
   unsigned char digits[EXPEVAL_DECIMAL_DIGITS];
   };

  /* Types and structures required to implement the consumber interface.
   * These structures and functions are provided to be used by the user
   * of the expression evaluation library.
   */

typedef char *(*expeval_callback_t)(expeval_context_t *context, void *parm);

  /* The result of evaluating an expression is to produce a value. Values
   * are also used to represent literals and the evaluation of variable
   * nodes also produces values. The expeval_value_t type is an abstraction
   * of the various values than could be involved in an expression
   * evaluation.
   */

typedef enum
   {
   EVAL_VALUE_UNKNOWN,       /* the type has not been set */
   EVAL_VALUE_NUMBER,        /* the value comprises of a number */
   EVAL_VALUE_STRING,        /* the value comprises of a string of chars */
   EVAL_VALUE_HEXSTR,        /* the value comprises of a string of hex chars */
   EVAL_VALUE_BOOLEAN        /* the value is either true or false */
   } value_type_t;

static char *type_names[] =
   { "Untyped", "Number", "String", "Hex String", "Boolean" };



struct value_string
   {
   int length;                   /* length of string */
   int memlen;                   /* length of the memory addressed */
   /* expeval_charset_t charset; */   /* encoding of string */
   unsigned char *string;        /* characters in string */
   };

struct value_number
   {
   EXPEVAL_NUMBER_TYPE number;   /* internal number format */
   };

struct expeval_value
   {
   expeval_value_t *link;   /* list of evaluated expressions */
   value_type_t type;       /* type of the value */

   int boolean;             /* the value if the type is EVAL_VALUE_BOOLEAN */
   value_string_t string;   /* the value if the type is EVAL_VALUE_STRING */
   value_number_t number;   /* the value if the type is EVAL_VALUE_NUMBER */
```

```
   };

struct expeval_variable
   {
   enum { EVAL_VAR_RESOLVED, EVAL_VAR_DEFER} status;
   char *name;             /* variable name */
   expeval_varcleanup_t cleanup;    /* optional variable cleanup function */
   expeval_varaddress_t varaddress; /* optional address function for compare */
   void *private_data;     /* variable library supplied data */
   expeval_context_t *context;      /* for getting the value of variable */
   /* value_type_t type; should refer to value.type */ /* type of the value */
   expeval_value_t value;  /* extracted variable value */

   void *resolve_private_data;     /* private data for var resolver callback */
   expeval_varvalue_t varvalue;    /* function for extracting variable value */
   void *varvalue_private_data;    /* private data for varvalue callback */
   expeval_byteimage_t byteimage;  /* function for creating byte image */
   };

struct expeval_function
   {
   char *name;                     /* function name */
   expeval_funcleanup_t cleanup;   /* optional function cleanup function */
   expeval_entrypoint_t entry;     /* function entry point */
   void *private_data;             /* private data for function entry */
   value_type_t ret_type;          /* return type of the function */
   int arg_count;                  /* parameter count */
   value_type_t arg_types[1];      /* types for argument */
   };

  /* An expression is a recursive data structure of expression nodes. Each
   * node can either be a leaf node or it can be an interior node. A leaf
   * node refers to a literal or to a variable. An interior node refers to
   * a node which represents an operator and sub-expression nodes which
   * represent the operands to which the operator is to be applied. To
   * represent the sub-expression nodes a peer-child strcuture is chosen,
   * and hence the expression tree is not necessarily a binary tree.
   */

typedef enum
   {
   NODE_INTERIOR,              /* node is interior node */
   NODE_COMPARE,               /* node is neat edge compare node */
   NODE_LITERAL,               /* node is literal value leaf node */
   NODE_VARIABLE,              /* node is variable leaf node */
   NODE_CALLBACK,              /* node is call-back leaf node */
   NODE_FUNCTION,              /* node is a function node */
   NODE_IFELSE                 /* node is condtional eval node */
   } expeval_node_type_t;

typedef enum
   {  /* must be in same order as op_code table entries */
```

```
   OP_NONE,
   /* arithmetic op_codes */
   OP_ADD, OP_SUBTRACT, OP_MULTIPLY, OP_DIVIDE, OP_REMAINDER,
   OP_INT_DIV, OP_POWER, OP_MINUS,
   /* general op_codes */
   OP_MIN, OP_MAX, OP_CMIN, OP_CMAX, OP_TOTAL, OP_UNIQUE,
   /* string op_codes */
   OP_CONCAT, OP_MATCH,
   /* relational op_codes */
   OP_EQUAL, OP_NOTEQUAL, OP_GREATER, OP_GREATEREQUAL,
   OP_LESS, OP_LESSEQUAL, OP_COMPARE_EQ, OP_COMPARE_NEQ,
   /* boolean op_codes */
   OP_AND, OP_OR, OP_XOR, OP_NOT,
   /* conditional evaluation */
   OP_CHOOSE, OP_UNLESS, OP_IFELSE
   } expeval_op_code_t;

struct expeval_node
   {
   expeval_node_t *peer;               /* next peer of this node */
   expeval_node_type_t type;           /* evaluation node type */
   expeval_value_t value;              /* value of this nodes evaluation */

   unsigned long flags;                /* Flags for internal processing */

#define EXPEVAL_NODE_COMMON 0x80000000     /* can be in multiple expressions */

  /* nodes of type NODE_INTERIOR */
   expeval_op_code_t op_code;          /* op_code of this node */
   int expeval_count;                  /* number times operator applied */
   expeval_node_t *child;              /* first child of this node */
   void *context;                      /* required for operand eval */

  /* nodes of type NODE_VARIABLE */
   expeval_variable_t *variable;       /* variable of node */

  /* nodes of type NODE_CALLBACK */
   expeval_callback_t callback;        /* call back function to get value */
   void *callback_parm;                /* parameter for call back function */

  /* nodes of type NODE_FUNCTION */
   expeval_function_t *function;       /* resolved function structure node */

  /* general hook for use by the operator at the node */
   void *image_value;                  /* for operator use, literal image */
   int user_data_len;                  /* for operator use */
   int user_data_pos;                  /* for operator use */
   void *reg_exp;                      /* for operator use, regular expr */
   void *private_data;                 /* for operator use */
   };

  /* The structure expeval_context supplies a context within which expressions
```

```
   * can be evaluated. This context structure needs to supply the means
   * of evaluating variables in the given expressions and for saving state
   * information which might be required for multiple evaluations of an
   * expression (for example looking for unique column values or summing
   * over columns in a collection).
   *
   * The context is also the structure into which user supplied functions
   * are set which are called when variable structures are created or
   * when literal structures are created.  This allows the private_date in
   * the corresponding structure to be setup.
   */

typedef enum
   {
   EVAL_ASCII,                    /* ASCII character set encoding used */
   EVAL_EBCDIC                    /* EBCDIC character set encoding used */
   } expeval_charset_t;

struct expeval_context
   {
   unsigned long flags;           /* options flags */

#  define EVALFL_VERBOSE 0x80000000  /* print verbose progress messages */
#  define EVALFL_LAZY    0x40000000  /* defer resolution of variables */
#  define EVALFL_ASCII   0x20000000  /* assume underlying char set ascii */
#  define EVALFL_EBCDIC  0x10000000  /* assume underlying char set ebcdic */

   expeval_charset_t data_charset;   /* character set assumed for data */
   int buf_len;                      /* length of buffer in context */
   unsigned char *buf;               /* current buffer in context */
   void *decctx;                     /* arithmetic library context address */
   expeval_output_callback_t action_callback; /* callback function */
   void *action_callback_parm;       /* callback parameter */
   char last_error[EXPEVAL_ERROR_MAX];   /* last error detected */

  /* Constant values and nodes. These should not be changed
   * by the user of the evaluate module.
   */
   expeval_value_t value_zero;       /* numeric constant zero */
   expeval_value_t value_one;        /* numeric constant one */
   expeval_value_t value_true;       /* boolean constant true */
   expeval_value_t value_false;      /* boolean constant false */

   expeval_node_t *node_zero;        /* numeric node zero */
   expeval_node_t *node_one;         /* numeric node one */
   expeval_node_t *node_true;        /* boolean node true */
   expeval_node_t *node_false;       /* boolean node false */

   expeval_supplier_t *default_var_supplier;  /* default supplier interface */
   hashtab_t *suppliers;             /* hash table of known suppliers */
   hashtab_t *functions;             /* hash table of defined functions */
```

```
  /* Hook for extraction method being used:
   */
   void *value_hook;                 /* for use by value extraction methods */

  /* Hash of hashes for common access.
   */
   hashtab_t *private_data;          /* hash of private data areas. This allows
                                        different code paths of expeval (where
                                        only the context structure is common) to
                                        register an entry here pointing to a
                                        useful data area.
                                        The first example of this is for the
                                        built in functions intable(), lookup()
                                        and inlookup() (and multiple async
                                        instantiations of each, if for example
                                        the same lookup is used twice in one
                                        expression) to share the same hash table
                                        of keywords and/or values.
                                     */

  /* For random deviates - See Numerical Recipes */
   long idum;
   expeval_value_callback_t value_callback; /* callback function for returning
                                               keyword and value of all
                                               variables in an expression within
                                               expeval_evaluate */
   void *value_callback_user_data;    /* Pointer passed back to caller's
                                         value callback routine */
   };

  /*
   * Exported functions.
   */

  /* Function for initialising and destroying a context to work in. The
   * context must be passed back to all the other functions. A context
   * is also the structure that needs to be updated in order to provide
   * a data context for the expression evaluation. The flags value can
   * be reset using the expeval_context_flags function.
   */

  /* Function for initialising and destroying a context to work in. The
   * context must be passed back to all the other functions. A context
   * is also the structure that needs to be updated in order to provide
   * a data context for the expression evaluation.
   */

expeval_context_t *expeval_context_open(unsigned long flags,
      expeval_output_callback_t callback, void *callback_parm);


  /* Function expeval_context_close() frees the resources held by a context
```

```
   * and frees the context. The context is not available after this call.
   */

void expeval_context_close(expeval_context_t *context);

  /* Function expeval_context_flags() can be used to change the internal
   * options of an existing context. Checking to make sure that the flag
   * values make sense and that they are consistant must be done here.
   */

int expeval_context_flags(expeval_context_t *context, unsigned long flags);

  /*
   * Functions for converting literals into leaf nodes.
   */

  /* Function expeval_value_string() is passed a string as an null
   * terminated sequence of characters and it updates a value node of
   * type string. If the encoding of the data is not in the hosting
   * representation then the encoding of the string is converted to
   * the same representation that the data is stored in.
   *
   * The function returns zero for successful completion, otherwise -1
   * is returned and a message is placed in last_error.
   */

int expeval_value_string(expeval_context_t *context, unsigned char *string,
      expeval_value_t *value);

  /* Function expeval_value_homest() is passed a string as an null
   * terminated sequence of characters and updates a value node of
   * type string. The encoding of the string is expected to be the host
   * machine's encoding and no translation of the string takes place.
   *
   * The function returns zero for successful completion, otherwise -1
   * is returned and a message is placed in last_error.
   */

int expeval_value_homest(expeval_context_t *context, unsigned char *string,
      expeval_value_t *value);

  /* Function expeval_value_hexdata() is passed a string as a null terminated
   * sequence of charaters and it updates a value node of type hexadecimal
   * string. The encoding of the string is expected to be independent of the
   * host encoding.
   *
   * The function returns zero for successful completion, otherwise -1
   * is returned and a message is placed in last_error.
   */

int expeval_value_hexdata(expeval_context_t *context, unsigned char *string,
      expeval_value_t *value);
```

```
  /* Function expeval_value_number() returns the internal number representation
   * of a number. The string is expected to conform to a valid null
   * terminated string representing numbers. The internal representation
   * is returned from the function.
   *
   * The function returns zero for successful completion, otherwise -1
   * is returned and a message is placed in last_error.
   */

int expeval_value_number(expeval_context_t *context, unsigned char *number,
      expeval_value_t *value);

  /* Function expeval_value_boolean() takes a string representing a boolean
   * value and converts this into an internal representation of the boolean
   * value. The string is expected to be a proper representation of a boolean
   * value.
   *
   * The function returns zero for successful completion, otherwise -1
   * is returned and a message is placed in last_error.
   */

int expeval_value_boolean(expeval_context_t *context, unsigned char *boolean,
      expeval_value_t *value);

  /*
   * Functions for building expression tree nodes:
   */

  /* Function expeval_build_var_leaf() creates a leaf node representing
   * a variable.
   */

expeval_node_t *expeval_build_var_leaf(expeval_context_t *context, char *name);

  /* Function expeval_resolve_var_leaf() recursively resolves variable leaf
   * nodes by by calling the supplied variable resolution function (which
   * is supplied by the caller of this library). The variable resolution
   * function places a value extracting method into the variable node and
   * an optional private data field for this function.  If all variable in
   * the tree are resolved properly then 0 is retruned. Othewise -1 is
   * returned on the first error encountered.
   */

int expeval_resolve_var_leaf(expeval_context_t *context, expeval_node_t *node);

  /* Function expeval_resolve_variable() is used to find a variable supplier
   * in order to determined the variable resolution function that will resolve
   * the current variable. The process of doing this is to determine the
   * first node of the variable name and then to use the upper case of this
   * as a lookup into the hash table of variable resolution functions.
   * If there is no supplier of the specific name, then the default supplier
```

```
   * is used. It is an error if there is no default supplier and there is
   * no specific supplier.  If the variable supplier has a variable resolve
   * function then this called to resolve the variable reference.
   */

int expeval_resolve_variable(expeval_context_t *context,
            expeval_variable_t *variable);


  /* Function expeval_build_lit_leaf() builds a leaf node representing a
   * literal value.
   */

expeval_node_t *expeval_build_lit_leaf(expeval_context_t *context,
      expeval_value_t *value);

  /* Function expeval_build_callback() builds a leaf node that has its
   * value represented by what is returned from a callback function.
   * When these nodes are evaluated, the value is obtained by calling the
   * callback function using the supplied context and parameter value.
   */

expeval_node_t *expeval_build_callback(expeval_context_t *context,
      value_type_t value_type, expeval_callback_t callback,
      void *callback_parm);

  /* Function expeval_build_interior() builds an interior node representing
   * an operator to apply to the children nodes. The child parameter is the
   * first child on the child's peer chain.
   */

expeval_node_t *expeval_build_interior(expeval_context_t *context,
      expeval_op_code_t op_code, expeval_node_t *child);

  /* Function expeval_build_compare() builds an compare node representing
   * a comparrison between a variable and a literal. The compare can be
   * compare equal or not-equal. This is a speacial operator as it will
   * compare the storage images of the item against the image of the
   * literal after applying the type of the literal to the item.
   */

expeval_node_t *expeval_build_compare(expeval_context_t *context,
      expeval_op_code_t op_code, expeval_node_t *child);

  /* Function expeval_build_function() builds a function call interior node.
   * The node is built using the supplied parameters expression arguments.
   * The name of the function is expected to exist in the functions table
   * in the current context. If an error occurs a message is formatted into
   * last_error in the supplied context and NULL is returned from the function.
   * Otherwise the new node is returned.
   */
```

```
expeval_node_t *expeval_build_function(expeval_context_t *context, char *name,
      expeval_node_t *arguments);

  /* Function expeval_build_ifelse() builds a conditional evaluation node.
   * The node is built using the supplied parameteter expression arguments.
   * If an error occurs a message is formatted into last_error in the supplied
   * context and NULL is returned from the function.
   * Otherwise the new node is returned.
   */

expeval_node_t *expeval_build_ifelse(expeval_context_t *context,
            expeval_node_t *arguments);

  /*
   * Functions for evaluating expressions and extracting the resultant
   * value.
   */

  /* Function expeval_type_expression() walks the given expression tree
   * assiging types to each of the nodes. Types are assigned bottom up
   * considering the child nodes and the operators at the child nodes to
   * derive the type of a node.
   */

int expeval_type_expression(expeval_context_t *context,
      expeval_node_t *expression);

  /* Function expeval_prime() is called before an expression is evaluated
   * the first time. This function prepares the evluation of the expression
   * where the state of the expression tree is accumulated over many calls
   * to the expeval_evaluate() function. For example, a column sum node.
   */

int expeval_prime(expeval_context_t *context, expeval_node_t *expression);

  /* Function expeval_final() is called after a series of expeval_evaluate()
   * function calls and is used to produce the results of column nodes
   * where this state is auumulated over multiple calls to expeval_evaluate().
   */

int expeval_final(expeval_context_t *context, expeval_node_t *expression);

  /* Function expeval_reset() resets all state information in the given
   * tree preparing it for the next evaluation.
   */

int expeval_reset(expeval_context_t *context, expeval_node_t *expression);

  /* Function expeval_evaluate() takes an expression and evaluates it within a
   * given context. The value value in the expression node is updated with the
   * resultant expression value and the function returns 0 if the value was
   * successfully evaluated. Otherwise the function returns -1 and the value
```

```
   * should not be used. The context for the evaluation is provided by the
   * context structure. This in turn includes any context that might be
   * required to extract the value, for example from a variable in a buffer.
   */

int expeval_evaluate(expeval_context_t *context, expeval_node_t *expression);

  /* Function expeval_value_to_string() extracts a string representation
   * of a value. If an error occurs during the processing, in which case
   * the formatting cannot be completed, then the function returns -1 and
   * places a message in the last_error field of the supplied context.
   * Otherwise the function returns the length of the string placed in the
   * supplied string buffer.
   */

int expeval_value_to_string(expeval_context_t *context, expeval_value_t *value,
      int buf_len, char *buffer);

  /* Function expeval_value_to_int() converts a numeric or boolean value to
   * an integer. If the function fails to convert the value then an error is
   * placed in the last_error field of the supplied context; otherwise 0
   * is returned and the value is placed in the supplied integer paramater.
   */

int expeval_value_to_int(expeval_context_t *context, expeval_value_t *value,
      int *integer);

  /* Function expeval_compile() compiles an expression tree from a text
   * expression. The returned expression tree is an unprimed, untyped
   * expression tree representing the given text.
   */

expeval_node_t *expeval_compile(expeval_context_t *context,
      char *expression_string);

  /* Cleanup functions */

int expeval_cleanup(expeval_node_t *expression);

  /* Utility functions:
   */

void data2home(expeval_context_t *context, value_string_t *string);
void home2data(expeval_context_t *context, value_string_t *string);

  /* Function expeval_varsup_open() is the supplier interface structure
   * creation function. This function creates a supplier interface
   * structure and places this structure in the hash table of known
   * supplier interfaces.
   */

expeval_supplier_t *expeval_varsup_open(expeval_context_t *context,
```

```
      char *name, expeval_resolve_t resolve, expeval_byteimage_t byteimage,
      expeval_varvalue_t varvalue, void *resolve_private_data);

  /* Function expeval_varsup_close() is the supplier interface cleanup
   * function. It is called for each supplier interface that has been
   * registered during the life of an expression context. The user of
   * the library does not have to call this function to remove the
   * interface. This will be done automatically during the cleanup
   * of the expression evaluation context.
   */

int expeval_varsup_close(expeval_context_t *context,
      expeval_supplier_t *supplier_interface);

  /* Function expeval_varsup_insert_storage() inserts a string/storage
   * value into a variable. The value passed to the function is interpreted
   * as a memory and not as a null terminated string. If an error occurs,
   * then the function returns -1 with a message will be placed into
   * the last_error field of the supplied context. Otherwise 0 is returned.
   */

int expeval_varsup_insert_storage(expeval_context_t *context,
      expeval_variable_t *variable, int buf_len, unsigned char *buffer);

  /* Function expeval_varsup_insert_number() inserts a numeric value in to a
   * variable. The value passed to the function is given as a null
   * terminated string and is stored into to the node in an internal format.
   * If an error occurs, the function returns -1 with a message being placed
   * in the last_error field of the context. Otherwise 0 is returned.
   */

int expeval_varsup_insert_number(expeval_context_t *context,
      expeval_variable_t *variable, char *number);


  /* Function expeval_varsup_insert_boolean() inserts a boolean value
   * represented by a string with the value of either "0" or "1" into the
   * supplied variable. If an error occurs, the function returns -1 with a
   * message being placed in the last_error field of the context.
   * Otherwise 0 is returned.
   */

int expeval_varsup_insert_boolean(expeval_context_t *context,
      expeval_variable_t *variable, char *value);

  /* Function expeval_varsup_define_function() introduces a new function
   * with its entry point, result type and argument types. A function
   * structure is built and registered under the supplied name.
   */

expeval_function_t *expeval_varsup_define_function(
      expeval_context_t *context, char *name, expeval_entrypoint_t entry,
```

```
      expeval_funcleanup_t cleanup, void *private_data, value_type_t ret_type,
      int arg_count, ...);

  /* Function expeval_varsup_define_function_ta() introduces a new function
   * with its entry point, result type and argument types. A function
   * structure is built and registered under the supplied name. The
   * difference between expeval_varsup_define_function() and
   * expeval_varsup_define_function_ta() is that the last parameter of this
   * function is an array which is a consolidation of the var args of the
   * function expeval_varsup_define_function().
   */

expeval_function_t *expeval_varsup_define_function_ta(
      expeval_context_t *context, char *name, expeval_entrypoint_t entry,
      expeval_funcleanup_t cleanup, void *private_data, value_type_t ret_type,
      int arg_count, value_type_t arg_types[]);

  /* Function expeval_varsup_delete_function() removes a function definition
   * from the given context. The function structure is removed from the hash
   * table and the storage associated with the function definition is freed.
   */

int expeval_varsup_delete_function(expeval_context_t *context,
      expeval_function_t *function);


  /*
   * Semantic action functions for the parser.
   */

  /* Function expeval_action_node() builds an interior node and plugs the node
   * add the node to the child chain by adjoining it to its next peer. The
   * supplied opcode is assigned to the node.
   */

expeval_node_t *expeval_action_node(expeval_context_t *context,
      expeval_op_code_t op_code, expeval_node_t *left, expeval_node_t* right);

  /* Function expeval_action_node_unary() builds an interior node and plugs the
   * node with the child node right operand. The supplied opcode is assigned to
   * the node.
   */

expeval_node_t *expeval_action_node_unary(expeval_context_t *context,
      expeval_op_code_t op_code, expeval_node_t* right);

  /* Function expeval_action_leaf_number() builds a literal leaf node
   * representing a numeric literal.
   */

expeval_node_t *expeval_action_leaf_number(expeval_context_t *context,
      char *number);
```

```
  /* Function expeval_action_leaf_string() builds a literal leaf node
   * representing a string literal.
   */

expeval_node_t *expeval_action_leaf_string(expeval_context_t *context,
      char *string);

  /* Function expeval_action_leaf_hstring() builds a literal leaf node
   * representing a hexadecimal string literal.
   */

expeval_node_t *expeval_action_leaf_hstring(expeval_context_t *context,
      char *string);

  /* Function expeval_action_leaf_home_string() builds a literal leaf node
   * representing a string literal represented in the local character encoding.
   */

expeval_node_t *expeval_action_leaf_home_string(expeval_context_t *context,
      char *string);

  /* Function expeval_action_leaf_variable() builds a variable reference leaf
   * node representing a variable reference. The variable reference will be
   * resolved later.
   */

expeval_node_t *expeval_action_leaf_variable(expeval_context_t *context,
      char *name);

  /* Function expeval_action_unary_minus() builds a numeric negation node by
   * subtracting the expression from zero.
   */

expeval_node_t *expeval_action_unary_minus(expeval_context_t *context,
      expeval_node_t* right);

  /* Function expeval_action_unary_not() builds a logical negation node by
   * xoring the expression with true.
   */

expeval_node_t *expeval_action_unary_not(expeval_context_t *context,
      expeval_node_t *right);

  /* Function expeval_action_build_function() builds a function call node. This
   * node also has a reference to an identifier which needs to be resolved later
   * as well as a series of expressions referring to the actual parameters of
   * the function.
   */

expeval_node_t *expeval_action_build_function(expeval_context_t *context,
      char *name, expeval_node_t *parameters);
```

```
  /* Function expeval_action_dlm_join() puts together the delimitted pairs of
   * the qualifiers of a variable name.
   */

char *expeval_action_dlm_join(expeval_context_t *context, char *string1,
      char *string2, char delim);

  /* Function expeval_action_string_index() puts an indexed item back together
   * with its index reference in the form that a resolver is expected to see the
   * index in a variable name.
   */

char *expeval_action_string_index(expeval_context_t *context, char *identifier,
      char *index);

  /* Function expeval_action_semantic_errors will return the number of semantic
   * errors that the parser encountered.
   */

int expeval_action_semantic_errors(void);

  /* Function expeval_value_sortkey() flattens the supplied value into the
   * supplied buffer in a manner that preserves the order of the value in the
   * space of all values. The function returns the number of bytes inserted
   * into the buffer. If an error occurs, then -1 is returned and an
   * error message is formatted into the last_error field of the context.
   */

int expeval_value_sortkey(expeval_context_t *context,
      expeval_value_t *value, unsigned char *buffer);

  /* Function expeval_number_sortkey() flattens a number into a given buffer
   * and then returns the nunber of bytes placed into the buffer. If an error
   * occurs then -1 is return instead. On error, an error message is formatted
   * into the last_error field of the context.
   */

int expeval_number_sortkey(expeval_context_t *context,
      value_number_t number, unsigned char *buffer);

  /* Function expeval_string_sortkey() flattens a string into a given buffer
   * and then returns number of bytes inserted into the buffer. If an error
   * occurs then -1 is return instead. On error, an error message is
   * formatted into the last_error field of the context.
   */

int expeval_string_sortkey(expeval_context_t *context,
     value_string_t string, unsigned char *buffer);

  /* Function expeval_boolean_sortkey() inserts an ordering for a boolean
   * value into a supplied buffer. The value is suitable for sorting. The
```

```
   * function returns the number of bytes placed into the buffer. On error
   * -1 is returned and an error message is formatted into the last_error
   * field of the context.
   */

int expeval_boolean_sortkey(expeval_context_t *context,
      int boolean, unsigned char *buffer);

  /* Function expeval_register_value_callback() will register a call back to be
   * invoked during expeval_evaluate() as the evaluation resolves the value of a
   * variable. This callback will allow the caller of expeval_evaluate() to
   * harvest the names and values of any variable nodes that make up the
   * expression being evaluated.
   */

int expeval_register_value_callback(expeval_context_t *context, void *user_data,
      expeval_value_callback_t value_callback);

  /* Function expeval_unregister_value_callback() allows a user to unregister
   * the value callback routine. This means that a caller can set the value
   * callback to run only for a specific evaluate call.
   * E.g.
   * expeval_register_value_callback(...)
   * expeval_evaluate(...)
   * expeval_unregister_value_callback(...)
   * means that the the call back will only be active for the one evaluate call.
   */

int expeval_unregister_value_callback(expeval_context_t *context);

#endif /* EXPEVAL_H */
```

References

[1] Ravi Sethi Alfred V. Aho, Monica S. Lam and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, second edition, 2007.