GNU `libmatheval` manual

Manual edition 1.1.3
For GNU `libmatheval` version 1.1.3
Last updated 5 May 2006

Aleksandar B. Samardžić

License

1 Introduction

GNU `libmatheval` is library comprising several procedures that makes possible to create in-memory tree representation of mathematical functions over single or multiple variables and later use this representation to evaluate function for specified variable values, to create corresponding tree for function derivative over specified variable or to get back textual representation of in-memory tree.

This section discuss use of programming interface exposed by library from C programs. Readers interested in Fortran interface should switch immediately to Chapter 3 [Fortran interface], page 13 section.

In order to use GNU `libmatheval` library from C code, it is necessary first to include header file '`matheval.h`' from all files calling GNU `libmatheval` procedures and then refer to '`libmatheval`' library among other linker option. Thus, command to compile C program using library and stored in file '`example.c`' using GNU C compiler would look like (supposing that library is installed using default prefix `/usr/local/lib`):

```
gcc example.c -I/usr/local/include -L/usr/local/lib -lmatheval -o exa
```

First step in actually utilizing library after including appropriate header file would be to declare variable of `void *` type to point to evaluator object that will represent given mathematical function:

```
void *f;
```

Then, given that textual representation of function is stored into string `buffer`, evaluator object corresponding to given mathematical function could be created using `evaluator_create()` procedure (see Section 2.1.1 [evaluator_create], page 6) as follows (see documentation for this procedure also for description of notation that should be used to describe mathematical functions):

```
f = evaluator_create (buffer); assert (f);
```

Return value should be always checked, because above procedure will return null pointer if there exist syntax errors in notation. After that, one could utilize `evaluator_get_variables()` (see Section 2.1.5 [evaluator_get_variables], page 8) procedure to obtain a list of variable names appearing in function:

```
{
  char **names;
  int count;
  int i;

  evaluator_get_variables (f, &names, &count);
  for (i = 0; i < count; i++)
    printf ("%s ", names[i]);
  printf ("\n");
}
```

Procedure `evaluator_evaluate()` (see Section 2.1.3 [evaluator_evaluate], page 7) could be used to evaluate function for specific variable values. Say that above function is over variable "x" only, then following code will evaluate and print function value for x = 0.1:

```
{
  char *names[] = { "x" };
  double values[] = { 0.1 };

  printf ("f(0.1) = %g\n", evaluator_evaluate (f, 1, names,
                                               values));
}
```

Or alternatively, since function is over variable with standard name "x", convenience procedure `evaluator_evaluate_x()` (Section 2.2.1 [evaluator_evaluate_x], page 9) could be used to accomplish same by following:

```
printf ("f(0.1) = %g\n", evaluator_evaluate_x (f, 0.1));
```

Evaluator object for function derivative over some variable could be created from evaluator object for given function. In order to accomplish this, a declaration for derivative evaluator object should be added to variable declarations section:

```
void *f_prim;
```

After that (supposing that "x" is used as derivation variable), derivative evaluator object could be created using `evaluator_derivative()` procedure (see Section 2.1.6 [evaluator_derivative], page 9):

```
f_prim = evaluator_derivative (f, "x");
```

or alternatively using `evaluator_derivative_x()` convenience procedure (see Section 2.2.4 [evaluator_derivative_x], page 11):

```
f_prim = evaluator_derivative_x (f);
```

Derivative evaluator object could be used to evaluate derivative values or say textual representation of derivative could be written to standard output through utilizing `evaluator_get_string()` procedure (see Section 2.1.4 [evaluator_get_string], page 8) to get string representing given evaluator. Following code would accomplish this:

```
printf ("  f'(x) = %s\n", evaluator_get_string (f_prim));
```

All evaluator objects must be destroyed after finished with using them and `evaluator_destroy()` procedure (see Section 2.1.2 [evaluator_destroy], page 7) is intended for this:

```
evaluator_destroy (f);
evaluator_destroy (f_prim);
```

Here follows complete program connecting above fragments. Program read from standard input string representing function over variable "x", create evaluators for function and its first derivative, print textual representation of function derivative to standard output, then read value of variable

"x" and finally print to standard output values of function and its first derivative for given value of variable "x".

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <matheval.h>

/* Size of input buffer.  */
#define BUFFER_SIZE 256

/* Program is demonstrating use of GNU libmatheval library of procedu
   for evaluating mathematical functions.  */
int
main (int argc, char **argv)
{
  char buffer[BUFFER_SIZE]; /* Input buffer.  */
  int length; /* Length of above buffer. */
  void *f, *f_prim; /* Evaluators for function and function derivativ
  char **names; /* Function variables names. */
  int count; /* Number of function variables. */
  double x; /* Variable x value.  */
  int i; /* Loop counter. */

  /* Read function.  Function has to be over variable x, or result ma
     be undetermined.  Size of textual represenatation of function is
     bounded here to 256 characters, in real conditions one should
     probably use GNU readline() instead of fgets() to overcome this
     limit.  */
  printf ("f(x) = ");
  fgets (buffer, BUFFER_SIZE, stdin);
  length = strlen (buffer);
  if (length > 0 && buffer[length - 1] == '\n')
    buffer[length - 1] = '\0';

  /* Create evaluator for function.  */
  f = evaluator_create (buffer);
  assert (f);

  /* Print variable names appearing in function. */
  evaluator_get_variables (f, &names, &count);
  printf ("  ");
  for (i = 0; i < count; i++)
    printf ("%s ", names[i]);
  printf ("\n");
```

```
  /* Create evaluator for function derivative and print textual
     representation of derivative.  */
  f_prim = evaluator_derivative_x (f);
  printf ("  f'(x) = %s\n", evaluator_get_string (f_prim));

  /* Read variable x value.  */
  printf ("x = ");
  scanf ("%lf", &x);

  /* Calculate and print values of function and its derivative for gi
     value of x.  */
  printf ("  f(%g) = %g\n", x, evaluator_evaluate_x (f, x));
  printf ("  f'(%g) = %g\n", x, evaluator_evaluate_x (f_prim, x));

  /* Destroy evaluators.  */
  evaluator_destroy (f);
  evaluator_destroy (f_prim);

  exit (EXIT_SUCCESS);
}
```

Above example exercise most of library main procedures (see Section 2.1 [Main entry points], page 6), as well as some of convenience procedures (see Section 2.2 [Convenience procedures], page 9). For full documentation, see Chapter 2 [Reference], page 6.

2 Reference

This section documents procedures constituting GNU `libmatheval` library. The convention is that all procedures have `evaluator_` prefix.

2.1 Main entry points

2.1.1 `evaluator_create()`

Synopsis

```
#include <matheval.h>

void *evaluator_create (char *string);
```

Description

Create evaluator object from `string` containing mathematical representation of function. Evaluator object could be used later to evaluate function for specific variable values or to calculate function derivative over some variable.

String representation of function is allowed to consist of decimal numbers, constants, variables, elementary functions, unary and binary operations.

Supported constants are (names that should be used are given in parenthesis): e (`e`), log2(e) (`log2e`), log10(e) (`log10e`), ln(2) (`ln2`), ln(10) (`ln10`), pi (`pi`), pi / 2 (`pi_2`), pi / 4 (`pi_4`), 1 / pi (`1_pi`), 2 / pi (`2_pi`), 2 / sqrt(pi) (`2_sqrtpi`), sqrt(2) (`sqrt`) and sqrt(1 / 2) (`sqrt1_2`).

Variable name is any combination of alphanumericals and `_` characters beginning with a non-digit that is not elementary function name.

Supported elementary functions are (names that should be used are given in parenthesis): exponential (`exp`), logarithmic (`log`), square root (`sqrt`), sine (`sin`), cosine (`cos`), tangent (`tan`), cotangent (`cot`), secant (`sec`), cosecant (`csc`), inverse sine (`asin`), inverse cosine (`acos`), inverse tangent (`atan`), inverse cotangent (`acot`), inverse secant (`asec`), inverse cosecant (`acsc`), hyperbolic sine (`sinh`), cosine (`cosh`), hyperbolic tangent (`tanh`), hyperbolic cotangent (`coth`), hyperbolic secant (`sech`), hyperbolic cosecant (`csch`), hyperbolic inverse sine (`asinh`), hyperbolic inverse cosine (`acosh`), hyperbolic inverse tangent (`atanh`), hyperbolic inverse cotangent (`acoth`), hyperbolic inverse secant (`asech`), hyperbolic inverse cosecant (`acsch`), absolute value (`abs`), Heaviside step function (`step`) with value 1 defined for x = 0 and Dirac delta function with infinity (`delta`) and not-a-number (`nandelta`) values defined for x = 0.

Supported unary operation is unary minus (`'-'`).

Supported binary operations are addition (`'+'`), subtraction (`'+'`), multiplication (`'*'`), division multiplication (`'/'`) and exponentiation (`'^'`).

Usual mathematical rules regarding operation precedence apply. Parenthesis (`'('` and `')'`) could be used to change priority order.

Blanks and tab characters are allowed in string representing function; newline characters must not appear in this string.

Return value

Pointer to evaluator object if operation successful, null pointer otherwise. Evaluator object is opaque, one should only use return pointer to pass it to other functions from library.

See also

Section 2.1.2 [evaluator_destroy], page 7, Section 2.1.3 [evaluator_evaluate], page 7, Section 2.1.4 [evaluator_get_string], page 8, Section 2.1.5 [evaluator_get_variables], page 8, Section 2.1.6 [evaluator_derivative], page 9

2.1.2 `evaluator_destroy()`

Synopsis

```
#include <matheval.h>

void evaluator_destroy (void *evaluator);
```

Description

Destroy evaluator object pointer by `evaluator` pointer. After returning from this call `evaluator` pointer must not be dereferenced because evaluator object gets invalidated.

Return value

None.

See also

Section 2.1.1 [evaluator_create], page 6

2.1.3 `evaluator_evaluate()`

Synopsis

```
#include <matheval.h>

double evaluator_evaluate (void *evaluator, int count, char **names,
    double *values);
```

Description

Calculate value of function represented by evaluator object for given variable values. Evaluator object is pointed by `evaluator` pointer. Variable names and corresponding values are given by `names` and `values` array respectively. Length of arrays is given by `count` argument.

Return value

Function value for given variable values. If some variable that appears in function is not mentioned in arguments, result is indeterminate. If all variables that appear in function are given, presence of variable or variables that doesn't appear in function in arguments has no effect, i.e. result is still exact.

See also

Section 2.1.1 [evaluator_create], page 6, Section 2.1.2 [evaluator_destroy], page 7, Section 2.2.1 [evaluator_evaluate_x], page 9, Section 2.2.2 [evaluator_evaluate_x_y], page 10, Section 2.2.3 [evaluator_evaluate_x_y_z], page 10

2.1.4 `evaluator_get_string()`

Synopsis

```
#include <matheval.h>

char *evaluator_get_string (void *evaluator);
```

Description

Return textual representation (i.e. mathematical function) of evaluator object pointed by `evaluator`. For notation used, see Section 2.1.1 [evaluator_create], page 6 documentation.

Return value

String with textual representation of evaluator object. This string is stored in evaluator object and caller must not free pointer returned by this function. Returned string is valid until evaluator object destroyed.

See also

Section 2.1.1 [evaluator_create], page 6, Section 2.1.2 [evaluator_destroy], page 7, Section 2.1.5 [evaluator_get_variables], page 8

2.1.5 `evaluator_get_variables()`

Synopsis

```
#include <matheval.h>

void evaluator_get_variables (void *evaluator, char ***names, int *co
```

Description

Return array of strings with names of variables appearing in function represented by evaluator. Address of array first element is stored by function in

location pointed by second argument and number of array elements is stored in location pointed by third argument. Array with function variable names is stored in evaluator object and caller must not free any of strings returned by this function nor array itself. Returned values are valid until evaluator object destroyed.

Return value

None.

See also

Section 2.1.1 [evaluator_create], page 6, Section 2.1.2 [evaluator_destroy], page 7, Section 2.1.4 [evaluator_get_string], page 8

2.1.6 `evaluator_derivative()`

Synopsis

```
#include <matheval.h>

void *evaluator_derivative (void *evaluator, char *name);
```

Description

Create evaluator for derivative of function represented by given evaluator object. Evaluator object is pointed to by `evaluator` pointer and derivation variable is determined by `name` argument. Calculated derivative is in mathematical sense correct no matters of fact that derivation variable appears or not in function represented by evaluator.

Return value

Pointer to evaluator object representing derivative of given function.

See also

Section 2.1.1 [evaluator_create], page 6, Section 2.1.2 [evaluator_destroy], page 7, Section 2.2.4 [evaluator_derivative_x], page 11, Section 2.2.5 [evaluator_derivative_y], page 12, Section 2.2.6 [evaluator_derivative_z], page 12

2.2 Convenience procedures

2.2.1 `evaluator_evaluate_x()`

Synopsis

```
#include <matheval.h>

double evaluator_evaluate_x (void *evaluator, double x);
```

Description

Convenience function to evaluate function for given variable "x" value. Function is equivalent to following:

```
char *names[] = { "x" };
double values[] = { x };

evaluator_evaluate (evaluator, sizeof (names) / sizeof(names[0]),
     names, values);
```

See Section 2.1.3 [evaluator_evaluate], page 7 for further information.

Return value

Value of function for given value of variable "x".

See also

Section 2.1.1 [evaluator_create], page 6, Section 2.1.2 [evaluator_destroy], page 7, Section 2.1.3 [evaluator_evaluate], page 7

2.2.2 evaluator_evaluate_x_y()

Synopsis

```
#include <matheval.h>

double evaluator_evaluate_x_y (void *evaluator, double x, double y);
```

Description

Convenience function to evaluate function for given variables "x" and "y" values. Function is equivalent to following:

```
char *names[] = { "x", "y" };
double values[] = { x, y };

evaluator_evaluate (evaluator, sizeof (names) / sizeof(names[0]),
     names, values);
```

See Section 2.1.3 [evaluator_evaluate], page 7 for further information.

Return value

Value of function for given values of variables "x" and "y".

See also

Section 2.1.1 [evaluator_create], page 6, Section 2.1.2 [evaluator_destroy], page 7, Section 2.1.3 [evaluator_evaluate], page 7

2.2.3 evaluator_evaluate_x_y_z()

Synopsis

```
#include <matheval.h>

double evaluator_evaluate_x_y_z (void *evaluator, double x, double y,
  double z);
```

Description

Convenience function to evaluate function for given variables "x", "y" and "z" values. Function is equivalent to following:

```
char *names[] = { "x", "y", "z" };
double values[] = { x, y, z };

evaluator_evaluate (evaluator, sizeof (names) / sizeof(names[0]),
     names, values);
```

See Section 2.1.3 [evaluator_evaluate], page 7 for further information.

Return value

Value of function for given values of variables "x", "y" and "z".

See also

Section 2.1.1 [evaluator_create], page 6, Section 2.1.2 [evaluator_destroy], page 7, Section 2.1.3 [evaluator_evaluate], page 7

2.2.4 evaluator_derivative_x()

Synopsis

```
#include <matheval.h>

void *evaluator_derivative_x (void *evaluator);
```

Description

Convenience function to differentiate function using "x" as derivation variable. Function is equivalent to:

```
evaluator_derivative (evaluator, "x");
```

See Section 2.1.6 [evaluator_derivative], page 9 for further information.

Return value

Evaluator object representing derivative of function over variable "x".

See also

Section 2.1.1 [evaluator_create], page 6, Section 2.1.2 [evaluator_destroy], page 7, Section 2.1.6 [evaluator_derivative], page 9

2.2.5 `evaluator_derivative_y()`

Synopsis

```
#include <matheval.h>

void *evaluator_derivative_y (void *evaluator);
```

Description

Convenience function to differentiate function using "y" as derivation variable. Function is equivalent to:

```
evaluator_derivative (evaluator, "y");
```

See Section 2.1.6 [evaluator_derivative], page 9 for further information.

Return value

Evaluator object representing derivative of function over variable "y".

See also

Section 2.1.1 [evaluator_create], page 6, Section 2.1.2 [evaluator_destroy], page 7, Section 2.1.6 [evaluator_derivative], page 9

2.2.6 `evaluator_derivative_z()`

Synopsis

```
#include <matheval.h>

void *evaluator_derivative_z (void *evaluator);
```

Description

Convenience function to differentiate function using "z" as derivation variable. Function is equivalent to:

```
evaluator_derivative (evaluator, "z");
```

See Section 2.1.6 [evaluator_derivative], page 9 for further information.

Return value

Evaluator object representing derivative of function over variable "z".

See also

Section 2.1.1 [evaluator_create], page 6, Section 2.1.2 [evaluator_destroy], page 7, Section 2.1.6 [evaluator_derivative], page 9

3 Fortran interface

Fortran interface to GNU `libmatheval` library is very similar to C interface; still, complete documentation from Chapter 2 [Reference], page 6 is reproduced here using Fortran terms in order to have Fortran programmer not to mess with C terms that he may not understand. Besides documentation for all library exported procedures, an example Fortran program of structure similar to sequence of code fragments presented for C programmers in Chapter 1 [Introduction], page 2 section as well as notes on how to link library with Fortran programs are presented here.

Since passing arguments between C and Fortran is not (yet) standardized, Fortran interface of library applies only to GNU Fortran 77 compiler; but note that same interface is working fine for GNU Fortran 95 compiler. Requests to adapt interface to other Fortran compilers are welcome (see section Chapter 5 [Bugs], page 26 for contact information), under condition that access to corresponding compiler is provided.

3.1 Fortran main entry points

3.1.1 `evaluator_create()`

Synopsis

```
integer*8 function evaluator_create (string)
  character(len=*) :: string
end function evaluator_create
```

Description

Create evaluator object from `string` containing mathematical representation of function. Evaluator object could be used later to evaluate function for specific variable values or to calculate function derivative over some variable.

String representation of function is allowed to consist of decimal numbers, constants, variables, elementary functions, unary and binary operations.

Supported constants are (names that should be used are given in parenthesis): e (`e`), log2(e) (`log2e`), log10(e) (`log10e`), ln(2) (`ln2`), ln(10) (`ln10`), pi (`pi`), pi / 2 (`pi_2`), pi / 4 (`pi_4`), 1 / pi (`1_pi`), 2 / pi (`2_pi`), 2 / sqrt(pi) (`2_sqrtpi`), sqrt(2) (`sqrt`) and sqrt(1 / 2) (`sqrt1_2`).

Variable name is any combination of alphanumericals and `_` characters beginning with a non-digit that is not elementary function name.

Supported elementary functions are (names that should be used are given in parenthesis): exponential (`exp`), logarithmic (`log`), square root (`sqrt`), sine (`sin`), cosine (`cos`), tangent (`tan`), cotangent (`cot`), secant (`sec`), cosecant (`csc`), inverse sine (`asin`), inverse cosine (`acos`), inverse tangent (`atan`), inverse cotangent (`acot`), inverse secant (`asec`), inverse cosecant (`acsc`), hyperbolic sine (`sinh`), cosine (`cosh`), hyperbolic tangent (`tanh`), hyperbolic

cotangent (`coth`), hyperbolic secant (`sech`), hyperbolic cosecant (`csch`), hyperbolic inverse sine (`asinh`), hyperbolic inverse cosine (`acosh`), hyperbolic inverse tangent (`atanh`), hyperbolic inverse cotangent (`acoth`), hyperbolic inverse secant (`asech`), hyperbolic inverse cosecant (`acsch`), absolute value (`abs`), Heaviside step function (`step`) with value 1 defined for x = 0 and Dirac delta function with infinity (`delta`) and not-a-number (`nandelta`) values defined for x = 0.

Supported unary operation is unary minus (`'-'`).

Supported binary operations are addition (`'+'`), subtraction (`'+'`), multiplication (`'*'`), division multiplication (`'/'`) and exponentiation (`'^'`).

Usual mathematical rules regarding operation precedence apply. Parenthesis (`'('` and `')'`) could be used to change priority order.

Blanks and tab characters are allowed in string representing function; newline characters must not appear in this string.

Return value

Positive 64-bit integer representing evaluator object unique handle if operation successful, 0 otherwise. Return value should be used only to pass it to other functions from library.

See also

Section 3.1.2 [Fortran evaluator_destroy], page 14, Section 3.1.3 [Fortran evaluator_evaluate], page 15, Section 3.1.4 [Fortran evaluator_get_string_length], page 15, Section 3.1.5 [Fortran evaluator_get_string_chars], page 16, Section 3.1.6 [Fortran evaluator_get_variables_length], page 16, Section 3.1.7 [Fortran evaluator_get_variables_chars], page 17, Section 3.1.8 [Fortran evaluator_derivative], page 17

3.1.2 `evaluator_destroy()`

Synopsis

```
subroutine evaluator_destroy (evaluator)
  integer*8 :: evaluator
end subroutine evaluator_destroy
```

Description

Destroy evaluator object denoted by `evaluator` handle. After returning from this call evaluator object gets invalidated, so value of `evaluator` handle should not be used any more.

Return value

None.

See also

Section 3.1.1 [Fortran evaluator_create], page 13

3.1.3 evaluator_evaluate()

Synopsis

```
double precision function evaluator_evaluate (evaluator, count, names
  integer*8 :: evaluator
  integer :: count
  character(len=*) :: names
  double precision :: values
  dimension values(*)
end function evaluator_evaluate
```

Description

Calculate value of function represented by evaluator object for given variable values. Evaluator object is identified by `evaluator` handle. Variable names are given by `names` string and corresponding values are given by `values` array respectively. Number of variables is given by `count` argument. Variable names in `names` string should be delimited by one or more blank characters.

Return value

Function value for given variable values. If some variable that appears in function is not mentioned in arguments, result is indeterminate. If all variables that appear in function are given, presence of variable or variables that doesn't appear in function in arguments has no effect, i.e. result is still exact.

See also

Section 3.1.1 [Fortran evaluator_create], page 13, Section 3.1.2 [Fortran evaluator_destroy], page 14, Section 3.2.1 [Fortran evaluator_evaluate_x], page 18, Section 3.2.2 [Fortran evaluator_evaluate_x_y], page 18, Section 3.2.3 [Fortran evaluator_evaluate_x_y_z], page 19

3.1.4 evaluator_get_string_length()

Synopsis

```
integer function evaluator_get_string_length (evaluator)
  integer*8 :: evaluator
end function evaluator_get_string_length
```

Description

Return length of textual representation (i.e. mathematical function) of evaluator object pointed by `evaluator`.

Return value

Evaluator textual representation string length.

See also

Section 3.1.1 [Fortran evaluator_create], page 13, Section 3.1.2 [Fortran evaluator_destroy], page 14, Section 3.1.5 [Fortran evaluator_get_string_chars], page 16

3.1.5 evaluator_get_string_chars()

Synopsis

```
subroutine evaluator_get_string_chars (evaluator)
  integer*8 :: evaluator
  character(len=*) :: string
end subroutine evaluator_get_string_chars
```

Description

Write textual representation (i.e. mathematical function) of evaluator object pointed by `evaluator` to string specified. For notation used, see Section 3.1.1 [Fortran evaluator_create], page 13 documentation. In order to declare string of appropriate length to be passed to this function, Section 3.1.4 [Fortran evaluator_get_string_length], page 15 function should be utilized.

Return value

None.

See also

Section 3.1.1 [Fortran evaluator_create], page 13, Section 3.1.2 [Fortran evaluator_destroy], page 14, Section 3.1.4 [Fortran evaluator_get_string_length], page 15

3.1.6 evaluator_get_variables_length()

Synopsis

```
integer function evaluator_get_variables_length (evaluator)
  integer*8 :: evaluator
end function evaluator_get_variables_length
```

Description

Return length of string with names of all variables (separated by a blank character) appearing in evaluator object pointed by `evaluator`.

Return value

Variable names string length.

See also

Section 3.1.1 [Fortran evaluator_create], page 13, Section 3.1.2 [Fortran evaluator_destroy], page 14, Section 3.1.7 [Fortran evaluator_get_variables_chars], page 17

3.1.7 `evaluator_get_variables_chars()`

Synopsis

```
subroutine evaluator_get_variables_chars (evaluator)
  integer*8 :: evaluator
  character(len=*) :: string
end subroutine evaluator_get_variables_chars
```

Description

Write names of all variables appearing in evaluator object pointed by `evaluator` into given string (separated by a blank character). In order to declare string of appropriate length to be passed to this function, Section 3.1.6 [Fortran evaluator_get_variables_length], page 16 function should be utilized.

Return value

None.

See also

Section 3.1.1 [Fortran evaluator_create], page 13, Section 3.1.2 [Fortran evaluator_destroy], page 14, Section 3.1.6 [Fortran evaluator_get_variables_length], page 16

3.1.8 `evaluator_derivative()`

Synopsis

```
integer*8 function evaluator_derivative (evaluator, name)
  integer*8 :: evaluator
  character(len=*) ::  name
end function evaluator_derivative
```

Description

Create evaluator for derivative of function represented by given evaluator object. Evaluator object is identified by `evaluator` handle and derivation variable is determined by `name` argument. Calculated derivative is in mathematical sense correct no matters of fact that derivation variable appears or not in function represented by evaluator.

Return value

64-bit integer uniquely identifying evaluator object representing derivative of given function.

See also

Section 3.1.1 [Fortran evaluator_create], page 13, Section 3.1.2 [Fortran evaluator_destroy], page 14, Section 3.2.4 [Fortran evaluator_derivative_x], page 19, Section 3.2.5 [Fortran evaluator_derivative_y], page 20, Section 3.2.6 [Fortran evaluator_derivative_z], page 20

3.2 Fortran convenience procedures

3.2.1 `evaluator_evaluate_x()`

Synopsis

```
double precision function evaluator_evaluate_x (evaluator, x)
  integer*8 :: evaluator
  double precision ::  x
end function evaluator_evaluate_x
```

Description

Convenience function to evaluate function for given variable "x" value. Function is equivalent to following:

```
evaluator_evaluate (evaluator, 1, 'x', (/ x /))
```

See Section 3.1.3 [Fortran evaluator_evaluate], page 15 for further information.

Return value

Value of function for given value of variable "x".

See also

Section 3.1.1 [Fortran evaluator_create], page 13, Section 3.1.2 [Fortran evaluator_destroy], page 14, Section 3.1.3 [Fortran evaluator_evaluate], page 15

3.2.2 `evaluator_evaluate_x_y()`

Synopsis

```
double precision function evaluator_evaluate_x_y (evaluator, x, y)
   integer*8 :: evaluator
   double precision ::  x, y
end function evaluator_evaluate_x_y
```

Description

Convenience function to evaluate function for given variables “x” and “y” values. Function is equivalent to following:

```
evaluator_evaluate (evaluator, 2, 'x y', (/ x, y /))
```

See Section 3.1.3 [Fortran evaluator_evaluate], page 15 for further information.

Return value

Value of function for given values of variables “x” and “y”.

See also

Section 3.1.1 [Fortran evaluator_create], page 13, Section 3.1.2 [Fortran evaluator_destroy], page 14, Section 3.1.3 [Fortran evaluator_evaluate], page 15

3.2.3 `evaluator_evaluate_x_y_z()`

Synopsis

```
double precision function evaluator_evaluate_x_y_z (evaluator, x, y,
  integer*8 :: evaluator
  double precision :: x, y, z
end function evaluator_evaluate_x_y_z
```

Description

Convenience function to evaluate function for given variables “x”, “y” and “z” values. Function is equivalent to following:

```
evaluator_evaluate (evaluator, 2, 'x y z', (/ x, y, z /))
```

See Section 3.1.3 [Fortran evaluator_evaluate], page 15 for further information.

Return value

Value of function for given values of variables “x”, “y” and “z”.

See also

Section 3.1.1 [Fortran evaluator_create], page 13, Section 3.1.2 [Fortran evaluator_destroy], page 14, Section 3.1.3 [Fortran evaluator_evaluate], page 15

3.2.4 `evaluator_derivative_x()`

Synopsis

```
integer*8 function evaluator_derivative_x (evaluator)
  integer*8 :: evaluator
end function evaluator_derivative_x
```

Description

Convenience function to differentiate function using “x” as derivation variable. Function is equivalent to:

```
evaluator_derivative (evaluator, 'x');
```

See Section 3.1.8 [Fortran evaluator_derivative], page 17 for further information.

Return value

Evaluator object representing derivative of function over variable “x”.

See also

Section 3.1.1 [Fortran evaluator_create], page 13, Section 3.1.2 [Fortran evaluator_destroy], page 14, Section 3.1.8 [Fortran evaluator_derivative], page 17

3.2.5 `evaluator_derivative_y()`

Synopsis

```
integer*8 function evaluator_derivative_y (evaluator)
  integer*8 :: evaluator
end function evaluator_derivative_y
```

Description

Convenience function to differentiate function using “y” as derivation variable. Function is equivalent to:

```
evaluator_derivative (evaluator, 'y');
```

See Section 3.1.8 [Fortran evaluator_derivative], page 17 for further information.

Return value

Evaluator object representing derivative of function over variable “y”.

See also

Section 3.1.1 [Fortran evaluator_create], page 13, Section 3.1.2 [Fortran evaluator_destroy], page 14, Section 3.1.8 [Fortran evaluator_derivative], page 17

3.2.6 `evaluator_derivative_z()`

Synopsis

```
integer*8 function evaluator_derivative_z (evaluator)
  integer*8 :: evaluator
end function evaluator_derivative_z
```

Description

Convenience function to differentiate function using "z" as derivation variable. Function is equivalent to:

```
evaluator_derivative (evaluator, 'z');
```

See Section 3.1.8 [Fortran evaluator_derivative], page 17 for further information.

Return value

Evaluator object representing derivative of function over variable "z".

See also

Section 3.1.1 [Fortran evaluator_create], page 13, Section 3.1.2 [Fortran evaluator_destroy], page 14, Section 3.1.8 [Fortran evaluator_derivative], page 17

3.3 Fortran sample program

Here follows sample program demonstrating use of library Fortran interface. Hopefully, comments throughout code will be enough for Fortran programmer to get acquainted with library usage. Basic functioning of program is equivalent to code presented for C programmer in Chapter 1 [Introduction], page 2 sequence, except that textual representation of function derivative is not printed to standard output and this is avoided simply because of Fortran 77 ugly string handling. Following code is written in Fortran 77 with GNU Fortran 77 compiler extensions (most notable of these certainly is free form of source code).

```
! Program is demonstrating use of GNU libmatheval library of procedur
! for evaluating mathematical functions.
program evaluator
  implicit none

  ! Declarations of GNU libmatheval procedures used.
  integer*8 evaluator_create
  integer*8 evaluator_derivative_x
  double precision evaluator_evaluate_x
  external evaluator_destroy

  ! Size of input buffer.
  integer :: BUFFER_SIZE
  parameter(BUFFER_SIZE = 256)

  character(len = BUFFER_SIZE) :: buffer ! Input buffer.
  integer*8 :: f, f_prim ! Evaluators for function and function deriv
  double precision :: x ! Variable x value.

  ! Read function.  Function has to be over variable x, or result may
```

```
  ! be undetermined.  Size of textual represenatation will be truncat
  ! here to BUFFER_SIZE characters, in real conditions one should
  ! probably come with something smarter to avoid this limit.
  write (*, '(A)') 'f(x) = '
  read (*, '(A)') buffer

  ! Create evaluator for function.
  f = evaluator_create (buffer);
  if (f == 0) stop

  ! Create evaluator for function derivative.
  f_prim = evaluator_derivative_x (f);
  if (f_prim == 0) stop

  ! Read variable x value.
  write (*, '(A)') 'x = '
  read (*, *) x

  ! Calculate and print values of function and its derivative for giv
  ! value of x.
  write (*,*) '  f (', x, ') = ', evaluator_evaluate_x (f, x)
  write (*,*) '  f'' (', x, ') = ', evaluator_evaluate_x (f_prim, x)

  ! Destroy evaluators.
  call evaluator_destroy (f)
  call evaluator_destroy (f_prim)
end program evaluator
```

3.4 Fortran build process

In order to be able to reference GNU `libmatheval` procedures from Fortran code, declarations of procedures that will be used should be repeated, like demonstrated by Section 3.3 [Fortran sample program], page 21 (once when interface upgraded to Fortran 90, modules and `use` statement will be employed here). Command for compilation Fortran program using library and stored in file '`example.f`' using GNU Fortran 77 compiler would look like (again supposing that library is installed using default prefix '`/usr/local/lib`'):

```
f77 example.f -ff90 -ffree-form -L/usr/local/lib -lmatheval -o exampl
```

4 Hacking

4.1 Design notes

As usual with an open source project, ultimate reference for anyone willing to hack on it is its source code. Every effort is put to have source code properly commented; having in mind that GNU `libmatheval` is rather simple project, it is reasonable to expect that this would be enough for anyone interested in project internals to get acquainted with it. Still, this section will briefly explain project design. See Section 4.2 [Project structure], page 24 section for description of where each functionality is located in source code.

Mathematical functions are represented as trees in computer memory. There are five different nodes in such a tree: number, constants, variables, functions, unary operations and binary operations. Single data structure is employed for tree nodes, while union is used over what is different among them. Numbers have unique value, unary and binary operations have unique pointer(s) to their operand(s) node(s). To represent constants, variables and functions, a symbol table is employed; thus constants, variables and functions have unique pointers to corresponding symbol table records (functions also have unique pointer to their argument node). All operations related to functions (e.g. evaluation or derivative calculation) are implemented as recursive operations on tree nodes. There exist a node operation that is not visible as external procedure and this is node simplification; this operation is very important regarding overall efficiency of other operations and is employed each time when new tree created.

Symbol table is implemented as hash table, where each bucket has linked list of records stored in it. Records store information of symbol name and type (variable or function), as well as some unique information related to evaluation: variable records store temporary variable value and function records store pointer to procedure used to actually calculate function value during evaluation. Hashing function described in *A.V. Aho, R. Sethi, J.D. Ullman, "Compilers - Principle, Techniques, and Tools", Addison-Wesley, 1986, pp 435-437* is used. Symbol tables are reference counted objects, i.e. could be shared.

Evaluator objects actually consists of function tree and reference to symbol table. Most of operations on evaluator objects are simply delegated to function tree root node.

For parsing strings representing mathematical functions, Lex and Yacc are employed. Scanner is creating symbol table records for variables, while for constants and functions it is only looking up existing symbol table records (before starting scanning, symbol table should be populated with records for constants and functions recognized by scanner). Parser is responsible for building function tree representation.

Couple error reporting procedures, as well as replacements for standard memory allocation routines are also present. These are rather standard

for all GNU projects, so deserve no further discussion. Further present in project are couple procedures for mathematical functions not implemented by C standard library, like cotangent, inverse cotangent and some hyperbolic and inverse hyperbolic functions.

Also present in project are stubs for Fortran code calling library. These stubs uses knowledge of GNU Fortran 77 compiler calling conventions, take parameters from Fortran 77 calls, eventually mangle them to satisfy primary C library interface and call library procedures to actually do the work, finally eventually mangling return values to satisfy Fortran 77 calling conventions again.

Most important thing to know before criticizing library design is that it is intentionally left as simple as it could be. Decision is now that eventual library usage should direct its improvements. Some obvious and intended improvements if enough interest for library arise are enumerated in Section 4.3 [Intended improvements], page 25 section. If having further suggestions, pleas see Chapter 5 [Bugs], page 26 sections for contact information.

4.2 Project structure

Interesting source files are mostly concentrated in '`lib`' subdirectory of distribution. Basic arrangement is rather standard for GNU projects, thus scanner is in '`scanner.l`' file, parser in '`parser.y`', error handling routines are in '`error.c`' and '`error.h`' files, replacements for standard memory allocation routines are in '`xmalloc.c`' and '`xmalloc.h`', additional mathematical functions are in '`xmath.c`' and '`xmath.c`'. Project specific files are: '`node.h`' and '`node.c`' files for tree representing mathematical function data structures and procedures, '`symbol_table.c`' and '`symbol_table.h`' for symbol table data structures and procedures and finally '`evaluator.c`' and '`matheval.h`' for evaluator object data structures and procedures (evaluator object data structure is moved to '`.c`' file because '`matheval.h`' is public header file and this data structure should be opaque). Fortran interface is implemented in '`f77_interface.c`' file.

File '`libmatheval.texi`' under '`doc`' subdirectory of distribution contains Texinfo source of project documentation (i.e. what you are reading now).

Subdirectory '`tests`' contains library test suite. Kind of mixed design is employed here - GNU autotest is used for test framework in order to achieve more portability, while number of small Guile scripts are performing tests. File '`matheval.c`' in '`tests`' subdirectory contains program extending Guile interpreter with GNU `libmatheval` procedures. Files with '`.at`' extension in same subdirectory in turn consist of fragments of Guile code that this extended Guile interpreter executes in order to conduct tests. File '`matheval.sh`' is shell wrapper for program contained in '`matheval.c`' file; this wrapper is used by autotest during testing instead of original program. Most interesting aspect of code from '`tests`' subdirectory is certainly Guile

interface for library that is implemented in '`matheval.c`' file; anyone intending to write more tests must before approaching this task become familiar with this interface.

4.3 Intended improvements

As stated in Section 4.1 [Design notes], page 23 section, GNU `libmatheval` is designed with intention to be simple and understandable and to eventually have its usage to govern improvements. Thus, further work will be primarily directed by user requests and of course, as usual with open source projects, with amount of spare time of primary developer (see Chapter 5 [Bugs], page 26 for contact information). However, there exist several obvious improvements that I'm willing to work on immediately if any interest of library arise and these are (in random order) listed below:

- Extend scanner to recognize more mathematical functions, to recognize alternative names for existing functions (e.g. to recognize both '`tg`' and '`tan`' as names for tangent function) and to recognize more constants.
- Implement variable hash table length for symbol table. As for now, hash table length is fixed to 211 that is reasonable for most cases, but it would certainly be more robust to have hash table to be constructed of length proportional say to length of string representing function.
- Add more simplifications to function tree representation. Only basic simplifications, mostly related to numbers subtrees consolidation and binary operations neutral elements are employed now. More ambitious optimization, using commutative, associative and distributive rules for binary operations would be desirable.
- Improve output when evaluator object is printed. Presently, parenthesis are always used around operations, while using them when necessary to establish proper evaluation priority order only would give prettier output
- Add more tests. Basic functionality of library is exercised through existing test suite, but present number of tests is certainly far from enough.
- Extend and improve error handling. There are couple `assert()`s left in code that may be replaced with some other mechanism, also probably error handling of more error conditions should be added to library.
- Add command line interface to library, i.e. write a program that will make possible to evaluate expression for given variable values where both specified in command line, as program arguments (for expressions without variables this program could be useful as a calculator).

There exists also an improvement that is obvious and necessary but because I'm not native speaker I'm unfortunately not able to accomplish it anything more than I already tried:

- Clean up English used in documentation.

5 Bugs

If you encounter something that you think is a bug, please report it immediately. Try to include a clear description of the undesired behavior. A test case that exhibits the bug or maybe even patch fixing it, would too be of course very useful.

Suggestions on improving library would be also more than welcome. Please see Chapter 4 [Hacking], page 23, for further information.

Please direct bug reports and eventual patches to `bug-libmatheval@gnu.org` mailing list. For suggestions regarding improvements and other `libmatheval` related conversation use author e-mail address `asamardzic@matf.bg.ac.yu`.

6 Rationale and history

The library is developed as a back-end for "Numerical Analysis" course taught during 1999/2000, 2000/2001 and 2001/2002 school years at Faculty of Mathematics, University of Belgrade. Most numerical libraries (library accompanying "Numerical Recipes" book most notably example) are asking programmer to write corresponding C code when it comes to evaluate mathematical functions. It seemed to me that it would be more appropriate (well, at least for above mentioned course) to have library that will make possible to specify functions as strings and then have them evaluated for given variable values, so I wrote first version of library during November 1999. Fortran interface is added to the library later; during January 2001 interface for Pacific Sierra VAST Fortran 90 translator was implemented and during September 2001 it was replaced by interface for Intel Fortran 90 compiler[1]. This library eventually went into rather stable state and was tested by number of other programs implementing various numerical methods and developed for the same course.

After completing engagement with this course, I thought it may be interesting for someone else to use this code and decided to make it publicly available. So, having some spare time during June 2002, I re-wrote whole library in preparation for public release, now employing simpler overall design and also using GNU auto-tools and what else was necessary according to GNU guidelines. The benefit is that final product looks much better now (well, at least to me and at least at the very moment of this writing), the drawback is that code is not thoroughly tested again. But certainly author would be more than happy to further improve and maintain it. Please see Chapter 5 [Bugs], page 26, for contact information.

The library source code was hosted on Savannah (`http://savannah.gnu.org/`) since Septembar 2002. In September 2003, library officially became part of GNU project.

[1] That was in turn replaced by interface for GNU Fortran 77 compiler in order to meet requirement that no GNU project should require use of non-free software

Index

Table of Contents