Arithmetic Expressions

`rs-expr` 2011-02-26

Ralph Schleicher

Table of Contents

1 Arithmetic Expressions **1**
1.1 Expression Objects 1
1.2 Parsing Expressions 1
1.3 Evaluating Expressions 1
1.4 Lexical Analyzer 2
1.4.1 Token Codes 2
1.4.2 Lexer Tuning 3
1.4.3 Existing Scanners 5
1.5 Symbolic Variables 6

Concept Index **8**

Symbol Index **9**

1 Arithmetic Expressions

The `rs-expr` library contains functions to read and evaluate arithmetic expressions at program run-time. All symbols described in this chapter are defined in the header file '`rs-expr.h`'.

1.1 Expression Objects

`void * rs_expr_new` (*void*) [Function]

Create an arithmetic expression.

Return value is the address of a new arithmetic expression object. In case of an error, a null pointer is returned and `errno` is set to describe the error.

`void rs_expr_delete` (*void* **`expr`*) [Function]

Delete an arithmetic expression.

Argument *expr* is the address of an arithmetic expression object. It is no error if the *expr* is a null pointer.

1.2 Parsing Expressions

`void * rs_expr_scan` (*void* **`expr`*, *char const* **`string`*, *char* ***`end`*) [Function]

Convert the initial part of *string* to an arithmetic expression.

First argument *expr* is an arithmetic expression object. The arithmetic expression is scanned and parsed in the scope of *expr*. If *expr* is a null pointer, a new arithmetic expression object is created. Otherwise, the previous arithmetic expression is replaced by the new arithmetic expression.

Any leading whitespace characters in *string* are discarded. Which characters are whitespace is determined by the `isspace` function.

If the third argument *end* is not a null pointer, a pointer to the tail of the string is stored in `*end`. Otherwise, it is an error if any non-whitespace character remains after the end of the arithmetic expression.

The normal return value is the address of the arithmetic expression object *expr*. If *string* is a null pointer, empty, contains only whitespace characters, does not contain an initial substring that has the expected syntax of an arithmetic expression, or if any other error occurs, no conversion is performed. In this case, `rs_expr_scan` returns a null pointer, the value stored in `*end` is the value of *string*, and the previous arithmetic expression stored in *expr* is no longer valid.

1.3 Evaluating Expressions

`double rs_expr_eval` (*void* **`expr`*) [Function]

Evaluate an arithmetic expression.

Argument *expr* is the address of an arithmetic expression object.

Return value is the numeric value of the current arithmetic expression stored in the arithmetic expression object *expr*. If *expr* is a null pointer or if no arithmetic expression is stored in *expr*, a value of zero is returned and `errno` is set to `EINVAL`.

1.4 Lexical Analyzer

The lexical analyzer (*lexer* for short) converts input data into tokens. This section documents the meaning of each token and how the user can adjust the lexer.

1.4.1 Token Codes

Each token has a unique token code. The lexer distinguishes between *punctuation tokens* and *word tokens*. A punctuation token is a printing character that is not an alphabetic character (a letter), a decimal digit, or a whitespace character. A word token is a sequence of one or more printing characters not beginning with a character of the punctuation token character set.

These symbolic constants of data type `int` are defined for punctuation tokens:

`RS_EXPR_POS`
: Unary plus operator; default `+`.

`RS_EXPR_NEG`
: Unary minus operator; default `-`.

`RS_EXPR_ADD`
: Plus operator; default `+`.

`RS_EXPR_SUB`
: Minus operator; default `-`.

`RS_EXPR_MUL`
: Multiplication operator; default `*`.

`RS_EXPR_DIV`
: Division operator; default `/`.

`RS_EXPR_POW`
: Exponentiation operator; default `^`.

`RS_EXPR_IF1`
`RS_EXPR_IF2`
: Conditional expression operators; default `?` and `:`.

`RS_EXPR_BEG`
: Beginning of group; default `(`.

`RS_EXPR_END`
: End of group; default `)`.

`RS_EXPR_SEP`
: List separator; default `,`.

These symbolic constants of data type `int` are defined for word tokens:

`RS_EXPR_NUM`
: Numeric constant.

`RS_EXPR_SYM`
`RS_EXPR_SY1`
: Symbolic constant.

`RS_EXPR_FUN`
: Unary function.

`RS_EXPR_BIN`
: Binary function.

`RS_EXPR_VAR`
: Symbolic variable.

1.4.2 Lexer Tuning

You can change the default behavior of the lexer to suite your needs.

`int rs_expr_control` (*void* `*expr`, *int* `command`, ...) [Function]

Perform an arithmetic expression control request command.

The number and data type of additional arguments and the meaning of the return value depends on the control request command *command*.

First argument *expr* is an arithmetic expression object. The following symbolic constants are defined for *command*:

`RS_EXPR_PUNCT_CHAR`

Set the character for the punctuation token *tok* to character code *c*.

Third argument *tok* (data type `int`) should be one out of the symbolic constants for punctuation tokens. See Section 1.4.1 [Token Codes], page 2.

Fourth argument *c* (data type `int`) should be a printing character that is not an alphabetic character (a letter), a decimal digit, or a whitespace character.

Return value is 0 on success. In case of an error, a value of -1 is returned and `errno` is set to describe the error.

You can disable individual operators by setting the character code *c* for that operator to 0.

`RS_EXPR_SCAN_NUM`

Use *func* for scanning numeric word tokens.

Third argument *func* is a pointer to a function that the lexer calls whenever it attempts to scan a numeric word token. You should define this function like:

`char * scan_num` (*char const* `*start`, *int* `*tok`, *double* `*val`) [Function]

First argument *start* is the current position in the input buffer; that is, the beginning of the numeric word token.

Return value is the buffer position at the end of the numeric word token; that is, the address of the first character not part of the numeric word token.

If the function succeeds, it should store the token code just scanned at the address where the second argument *tok* points to. The only valid token code is `RS_EXPR_NUM`. The numeric value itself has to be stored at the address where the third argument *val* points to.

If *start* does not point to a valid numeric word token, the return value should be *start*. In case of a fatal error, the return value should be a null pointer.

The `RS_EXPR_SCAN_NUM` control request returns 0 on success. In case of an error, a value of -1 is returned and `errno` is set to describe the error.

See Section 1.4.3 [Existing Scanners], page 5, for predefined scanners. You can disable scanning of numbers if you specify a null pointer as the third argument to the `RS_EXPR_SCAN_NUM` control request.

`RS_EXPR_SCAN_SYM`

Use *func* for scanning symbolic word tokens.

Third argument *func* is a pointer to a function that the lexer calls whenever it attempts to scan a symbolic word token. You should define this function like:

`char * scan_sym` (*char const* `*start`, *int* `*tok`, *void* `**ref`) [Function]

First argument *start* is the current position in the input buffer; that is, the beginning of the symbolic word token.

Return value is the buffer position at the end of the symbolic word token; that is, the address of the first character not part of the symbolic word token.

If the function *scan_sym* succeeds, it should store the token code just scanned at the address where the second argument *tok* points to and the address of an external symbol at the address where the third argument *ref* points to. The definition of the external symbol depends on the token code. Valid token codes, their meaning, and the corresponding definition of the external symbol are listed in the following table:

`RS_EXPR_SYM`
: Symbolic constant defined as `double` *`var`*.

`RS_EXPR_SY1`
: Symbolic constant defined as `double` *`func`* `(void)`.

`RS_EXPR_FUN`
: Unary function defined as `double` *`func`* `(double` *`x`*`)`.

`RS_EXPR_BIN`
: Binary function defined as `double` *`func`* `(double` *`x`*`, double` *`y`*`)`.

`RS_EXPR_VAR`
: Symbolic variable defined as `void *`*`var`*.

  You have to install additional call-back functions if you return this token code. See the `RS_EXPR_COMPAR_VAR`, `RS_EXPR_EVAL_VAR`, and `RS_EXPR_FREE_VAR` control request described below, for how to do that.

If *start* does not point to a valid symbolic word token, the return value should be *start*. In case of a fatal error, the return value should be null pointer.

The `RS_EXPR_SCAN_SYM` control request returns 0 on success. In case of an error, a value of -1 is returned and `errno` is set to describe the error.

See Section 1.4.3 [Existing Scanners], page 5, for predefined scanners. You can disable scanning of symbols if you specify a null pointer as the third argument to the `RS_EXPR_SCAN_SYM` function.

`RS_EXPR_COMPAR_VAR`

Third argument *func* is a pointer to a function that is called to compare two symbolic variables. It is a fatal error if an arithmetic expression contains symbolic variables and this call-back function is not set when an arithmetic expression is scanned. You should define this function like:

`int` ***`compar_var`*** (*void const* *`a`, *void const* *`b`) [Function]

Arguments *a* and *b* are symbolic variable references returned to the lexer by the scanner for symbolic word tokens.

Return value should be 0 if the two symbolic variables are equal. If *a* is considered less than *b*, the return value should be less than 0, and if *a* is considered greater than *b*, the return value should be greater than 0.

The `RS_EXPR_COMPAR_VAR` control request returns 0 on success. In case of an error, a value of -1 is returned and `errno` is set to describe the error.

`RS_EXPR_EVAL_VAR`

Third argument *func* is a pointer to a function that is called to evaluate a symbolic variable. It is a fatal error if an arithmetic expression contains symbolic variables and this call-back function is not set when the arithmetic expression is evaluated. You should define this function like:

`double` ***`eval_var`*** `(`*void *`var`*`)` [Function]

Argument *var* is the symbolic variable reference returned to the lexer by the scanner for symbolic word tokens.

Return value should be the value of the symbolic variable *var*.

The `RS_EXPR_EVAL_VAR` control request returns 0 on success. In case of an error, a value of -1 is returned and `errno` is set to describe the error.

`RS_EXPR_FREE_VAR`

Third argument *func* is a pointer to a function that is called to delete a symbolic variable. It is no error if *func* is a null pointer. You should define this function like:

`void` ***`free_var`*** `(`*void *`var`*`)` [Function]

Argument *var* is the symbolic variable reference returned to the lexer by the scanner for symbolic word tokens.

The `RS_EXPR_FREE_VAR` control request returns 0 on success. In case of an error, a value of -1 is returned and `errno` is set to describe the error.

1.4.3 Existing Scanners

This section documents the predefined scanners for numeric and symbolic word tokens. See Section 1.4.2 [Lexer Tuning], page 3, for how to replace a scanner.

`char *` **`rs_expr_scan_num`** `(`*char const *`start`, int *`tok`, double *`val`*`)` [Function]

The default scanner for numeric constants.

This scanner uses the `strtod` function to convert the initial part of *start* into a number.

Return value is the string position at the end of the number; that is, the address of the first character not part of the number. If no number is scanned, the return value is *start*. In case of an error, a null pointer is returned.

`char *` **`rs_expr_scan_sym`** `(`*char const *`start`, int *`tok`, void **`ref`*`)` [Function]

The default scanner for symbolic word tokens.

This scanner recognizes the following symbols:

`e`	Base of the natural logarithm.
`pi`	Ratio of a circle's circumference to its diameter.
`inf`	Positive infinity.
`nan`	Not-a-number.
`abs (`*x*`)`	Absolute value.

`ceil (`*x*`)`
`floor (`*x*`)`
`trunc (`*x*`)`
	Nearest integer functions.

`ceiling (`*x*`)`	Alternative name for the `ceil` function.

`sin (`*x*`)`
`cos (`*x*`)`
`tan (`*x*`)`
	Sine, cosine, and tangent function.

`arcsin (`*x*`)`
`arccos (`*x*`)`
`arctan (`*x*`)`
: Arc sine, arc cosine, and arc tangent function.

`asin (`*x*`)`
`acos (`*x*`)`
`atan (`*x*`)`
: Alternative names for the arc sine, arc cosine, and arc tangent function.

`exp (`*x*`)`
: Base e exponentiation.

`log (`*x*`)`
`ln (`*x*`)`
: Natural logarithm.

`pow (`*x*`, `*y*`)`
: General exponentiation.

`sqrt (`*x*`)`
`cbrt (`*x*`)`
: Square root and cube root function.

`sinh (`*x*`)`
`cosh (`*x*`)`
`tanh (`*x*`)`
: Hyperbolic sine, hyperbolic cosine, and hyperbolic tangent function.

`arsinh (`*x*`)`
`arcosh (`*x*`)`
`artanh (`*x*`)`
: Inverse hyperbolic sine, inverse hyperbolic cosine, and inverse hyperbolic tangent function.

`asinh (`*x*`)`
`acosh (`*x*`)`
`atanh (`*x*`)`
: Alternative names for the inverse hyperbolic sine, inverse hyperbolic cosine, and inverse hyperbolic tangent function.

`erf (`*x*`)`
`erfc (`*x*`)`
: Error function and complementary error function.

`gamma (`*x*`)`
: Gamma function.

`angle (`*x*`, `*y*`)`
: Signed angle of a point in the plane.

`hypot (`*x*`, `*y*`)`
: Distance of a point in the plane from the origin.

Return value is the string position at the end of the symbol; that is, the address of the first character not part of the symbol. If no symbol is scanned *start* is returned. In case of an error, a null pointer is returned.

1.5 Symbolic Variables

The default lexer (see Section 1.4.2 [Lexer Tuning], page 3, and see Section 1.4.3 [Existing Scanners], page 5, for more details) does not handle symbolic variables. If you install a user-defined scanner for symbolic variables, the following functions may help you to manage symbolic variables.

`void rs_expr_walk_var` (*void* `*expr`, *void* `*func`, *void* `*arg`) [Function]

Call function *func* for each symbolic variable defined in the scope of the arithmetic expression object *expr*.

If the second argument *arg* is a null pointer, you should define this function like:

`void walk_var` (*void* `*var`) [Function]

Argument *var* is the symbolic variable reference returned to the lexer by the scanner for symbolic word tokens.

Otherwise, if the second argument *arg* is not a null pointer, you should define this function like:

`void walk_var_2` (*void* `*var`, *void* `*arg`) [Function]

First argument *var* is the symbolic variable reference returned to the lexer by the scanner for symbolic word tokens.

Second argument *arg* is the third argument of the `rs_expr_walk_var` function.

Concept Index

B

beginning of group . . . 2
binary function . . . 2

C

closing parenthesis . . . 2
conditional operator . . . 2
constant, numeric . . . 2
constant, symbolic . . . 2

D

disabling numbers . . . 3
disabling operators . . . 3
disabling symbols . . . 4
division operator . . . 2

E

end of group . . . 2
exponentiation operator . . . 2

F

function, binary . . . 2
function, unary . . . 2

G

group, beginning . . . 2
group, end . . . 2

L

list separator . . . 2

M

minus operator . . . 2
multiplication operator . . . 2

N

numbers, disabling . . . 3
numeric constant . . . 2

O

opening parenthesis . . . 2
operator, conditional . . . 2
operator, division . . . 2
operator, exponentiation . . . 2
operator, minus . . . 2
operator, multiplication . . . 2
operator, plus . . . 2
operator, unary minus . . . 2
operator, unary plus . . . 2
operators, disabling . . . 3

P

parenthesis, closing . . . 2
parenthesis, opening . . . 2
plus operator . . . 2
punctuation token codes . . . 2
punctuation tokens . . . 2

S

separator, list . . . 2
symbolic constant . . . 2
symbolic variable . . . 2
symbols, disabling . . . 4

T

token codes, punctuation . . . 2
token codes, word . . . 2
tokens, punctuation . . . 2
tokens, word . . . 2

U

unary function . . . 2
unary minus operator . . . 2
unary plus operator . . . 2

V

variable, symbolic . . . 2

W

word token codes . . . 2
word tokens . . . 2

Symbol Index

C

compar_var 4

E

eval_var 5

F

free_var 5

R

RS_EXPR_ADD 2
RS_EXPR_BEG 2
RS_EXPR_BIN 2
RS_EXPR_COMPAR_VAR 4
rs_expr_control 3
rs_expr_delete 1
RS_EXPR_DIV 2
RS_EXPR_END 2
rs_expr_eval 1
RS_EXPR_EVAL_VAR 4
RS_EXPR_FREE_VAR 5
RS_EXPR_FUN 2
RS_EXPR_IF1 2
RS_EXPR_IF2 2
RS_EXPR_MUL 2
RS_EXPR_NEG 2
rs_expr_new 1
RS_EXPR_NUM 2
RS_EXPR_POS 2
RS_EXPR_POW 2
RS_EXPR_PUNCT_CHAR 3
rs_expr_scan 1
rs_expr_scan_num 5
RS_EXPR_SCAN_NUM 3
rs_expr_scan_sym 5
RS_EXPR_SCAN_SYM 3
RS_EXPR_SEP 2
RS_EXPR_SUB 2
RS_EXPR_SY1 2
RS_EXPR_SYM 2
RS_EXPR_VAR 2
rs_expr_walk_var 7

S

scan_num 3
scan_sym 3

W

walk_var 7
walk_var_2 7