Small, Fast S-Expression Library Reference Manual

Generated by Doxygen 1.4.6

Thu Oct 25 01:19:37 2007

Contents

Chapter 1

A small and quick S-expression parsing library.

1.1 Introduction

This library was created to provide s-expression parsing and manipulation facilities to C and C++ programs. The primary goals were speed and efficiency - low memory impact, and the highest speed we could achieve in parsing. Suprisingly, no other libraries on the net were found that were not bloated with features or involved embedding a full-fledged LISP or Scheme interpreter into our programs. So, this library evolved to fill this gap. As such, it does not guarantee that every valid LISP expression is parseable, and many features that are not required aren't implemented. See Rivest's S-expression library for an example of a much more featureful library.

What features does this library include? At the heart of the code is a continuation-based parser implementing a basic parser state machine. Continuations allow users to accumulate multiple streams of characters, and parse each stream simultaneously. A continuation allows the parser to stop midstream, start working on a new expression, and return to the first without corruption of complex state management in the users code. No threads, no forking, nothing more than a data structure that must be passed in and captured as data becomes available to parse. Once an expression has been parsed, a simple structure is returned that represents the "abstract syntax tree" of the parsed expression. For the majority of users, the parser and this data structure will be all that they will ever need to see. For convenience reasons, other functions such as I/O wrappers and AST traversal routines have been included, but they are not required if users don't wish to use them.

1.2 Credits

SFSEXP: Small, Fast S-Expression Library version 1.2, October 2007

Written by Matthew Sottile (matt@cs.uoregon.edu)

1.3 License Information

Chapter 2

Small, Fast S-Expression Library Module Index

2.1 Small, Fast S-Expression Library Modules

Here is a list of all modules:

Chapter 3

Small, Fast S-Expression Library Hierarchical Index

3.1 Small, Fast S-Expression Library Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Small, Fast S-Expression Library Data Structure Index

4.1 Small, Fast S-Expression Library Data Structures

Here are the data structures with brief descriptions:

Chapter 5

Small, Fast S-Expression Library File Index

5.1 Small, Fast S-Expression Library File List

Here is a list of all files with brief descriptions:

Chapter 6

Small, Fast S-Expression Library Module Documentation

6.1 I/O routines

Data Structures

- struct sexp_iowrap

Typedefs

- typedef sexp_iowrap sexp_iowrap_t

Functions

- sexp_iowrap_t * init_iowrap (int fd)
- void destroy_iowrap (sexp_iowrap_t *iow)
- sexp_t * read_one_sexp (sexp_iowrap_t *iow)

6.1.1 Typedef Documentation

6.1.1.1 typedef struct sexp_iowrap sexp_iowrap_t

This structure is a wrapper around a standard I/O file descriptor and the parsing infrastructure (continuation and a buffer) required to parse off of it. This is used so that routines can hide the loops and details required to accumulate up data read off of the file descriptor and parse expressions individually out of it.

6.1.2 Function Documentation

6.1.2.1 void destroy_iowrap (sexp_iowrap_t * *iow*)

destroy an IO wrapper structure. The file descriptor wrapped in the wrapper will **not** be closed, so the caller is responsible for manually calling close on the file descriptor.

6.1.2.2 sexp_iowrap_t* init_iowrap (int *fd*)

create an IO wrapper structure around a file descriptor. A NULL return value indicates some problem occurred allocating the wrapper, so the user should check the value of sexp_errno for further information.

6.1.2.3 sexp_t* read_one_sexp (sexp_iowrap_t * *iow*)

given and IO wrapper handle, read one s-expression off of it. this expression may be contained in a continuation, so there is no guarantee that under the covers an IO read actually is occuring. A return value of NULL can either indicate a parser error or no more data on the input IO handle. In the event that NULL is returned, the user should check to see if sexp_errno contains SEXP_ERR_IO_EMPTY (no more data) or a more problematic error.

6.2 Parser routines

Functions

- sexp_errcode_t set_parser_buffer_params (size_t ss, size_t gs)
- sexp_t * parse_sexp (char *s, size_t len)
- sexp_t * iparse_sexp (char *s, size_t len, pcont_t *cc)
- pcont_t * cparse_sexp (char *s, size_t len, pcont_t *pc)

6.2.1 Function Documentation

6.2.1.1 pcont_t* cparse_sexp (char * *s*, size_t *len*, pcont_t * *pc*)

given a LISP style s-expression string, parse it into a set of connected sexp_t structures.

6.2.1.2 sexp_t* iparse_sexp (char * *s*, size_t *len*, pcont_t * *cc*)

wrapper around parser for friendlier continuation use pre-condition : continuation (cc) is NON-NULL!

6.2.1.3 sexp_t* parse_sexp (char * *s*, size_t *len*)

wrapper around parser for compatibility.

6.2.1.4 sexp_errcode_t set_parser_buffer_params (size_t *ss*, size_t *gs*)

Set the parameters on atom value buffer allocation and growth sizes. This is an important point for performance tuning, as many factors in the expected expression structure must be taken into account such as:

- Average size of atom values
- Variance in sizes of atom values
- Amount of memory that is tolerably ”wasted” (allocated but not used)

The *ss* parameter specifies the initial size of all atom buffers. Ideally, this should be sufficiently large to capture MOST atom values, or at least close enough such that one growth is required. The *gs* parameter specifies the number of bytes to increase the buffer size by when space is exhausted. A safe choice for parameter sizes would be on the order of the average size for *ss*, and one standard deviation for *gs*. This ensures

that 50% of all expressions are guaranteed to fit in the initial buffer, and rougly 80-90% will fit in one growth. If memory is not an issue, choosing ss to be the mean plus one standard deviation will capture 80-90% of expressions in the initial buffer, and a gs of one standard deviation will capture nearly all expressions.

Note: These parameters can be tuned at runtime as needs change, and they will be applied to all expressions and expression elements parsed after they are modified. They will not be applied retroactively to expressions that have already been parsed.

6.3 Visualization and debugging routines

Functions

- sexp_errcode_t sexp_to_dotfile (const sexp_t *sx, const char *fname)

6.3.1 Function Documentation

6.3.1.1 sexp_errcode_t sexp_to_dotfile (const sexp_t * *sx*, const char * *fname*)

Given a s-expression and a filename, this routine creates a DOT-file that can be used to generate a visualization of the s-expression data structure. This is useful for debugging to ensure that the structure is correct and follows what was expected by the programmer. Non-trivial s-expressions can yield very large visualizations though. Sometimes it is more practical to visualize a portion of the structure if one knows where a bug is likely to occur.

Parameters:

sx S-expression data structure to create a DOT file based on.

fname Filename of the DOT file to emit.

Chapter 7

Small, Fast S-Expression Library Data Structure Documentation

7.1 __cstring Struct Reference

`#include <cstring.h>`

Data Fields

- char * base
- size_t len
- size_t curlen

7.1.1 Detailed Description

Structure wrapping the character pointer and size counters (allocated vs. actual used).

7.1.2 Field Documentation

7.1.2.1 char* base

Base address of the string.

7.1.2.2 size_t curlen

Current size of the string stored in the buffer. len >= curlen always, and when len < curlen would be true after a concat operation, we realloc bigger space to keep len >= curlen.

7.1.2.3 size_t len

Size of the memory allocated and pointed to by the base pointer.

7.2 elt Struct Reference

`#include <sexp.h>`

Data Fields

- elt_t ty
- char * val
- size_t val_allocated
- size_t val_used
- elt * list
- elt * next
- atom_t aty
- char * bindata
- size_t binlength

7.2.1 Detailed Description

An s-expression is represented as a linked structure of elements, where each element is either an *atom* or *list*. An atom corresponds to a string, while a list corresponds to an s-expression. The following grammar represents our definition of an s-expression:

```
sexpr  ::= ( sx )
sx     ::= atom sxtail | sexpr sxtail | 'sexpr sxtail | 'atom sxtail | NULL
sxtail ::= sx | NULL
atom   ::= quoted | value
quoted ::= "ws_string"
value  ::= nws_string
```

An atom can either be a quoted string, which is a string containing whitespace (possibly) surrounded by double quotes, or a non-whitespace string that does not require surrounding quotes. An element representing an atom will have a type of *value* and data stored in the *val* field. An element of type *list* represents an s-expression corresponding to *sexpr* in the grammar, and will have a pointer to the head of the appropriate s-expression. Details regarding these fields and their values given with the fields themselves. Notice that a single quote can appear directly before an s-expression or atom, similar to the use in LISP.

7.2.2 Field Documentation

7.2.2.1 atom_t aty

For elements that represent *values*, this field will specify the specific type of value that it represents. This can be used by functions to determine how this value should be printed (ie: how it should be quoted) or interpreted (ie: interpreting s-expressions that are prefixed with a tick-mark.). This value also indicates whether or not the programmer should look in the val field or bindata field for the atom data.

7.2.2.2 char* bindata

For elements that represent *binary* blobs, this field will point to a memory location where the data resides. The length of this memory blob is the next field. char* implies byte sized elements. This is only used in INLINE_BINARY parser mode. **IMPORTANT NOTE**: The data in this field is freed on a destroy_sexp() call, so users should copy it to memory they allocate if they wish it to persist after the sexp_t has been freed.

7.2.2.3 size_t binlength

The length of the data pointed at by bindata in bytes.

7.2.2.4 struct elt* list

If the type of the element is **SEXP_LIST**, this field will contain a pointer to the head element of the list.

7.2.2.5 struct elt* next

The *next* field is a pointer to the next element in the current expression. If this element is the last element in the s-expression, this field will be *NULL*.

7.2.2.6 elt_t ty

The element has a type that determines how the structure is used. If the type is **SEXP_VALUE**, then a programmer knows that either the val field or bindata field is meaningful dependin on the value of the aty field, and contains the data associated with this element of the s-expression. If the type is **SEXP_LIST**, then the list field contains a pointer to the s-expression element representing the head of the list. For each case, the field for the opposite case contains no meaningful data and using them in any way is likely to cause an error.

7.2.2.7 char* val

If the type of the element is **SEXP_VALUE** and the aty field is not **SEXP_BINARY**, this field will contain the actual data represented by this element.

7.2.2.8 size_t val_allocated

Number of bytes allocated for val.

7.2.2.9 size_t val_used

Number of bytes used in val (<= val_allocated).

7.3 parser_event_handlers Struct Reference

`#include <sexp.h>`

Data Fields

- void(∗ start_sexpr)()
- void(∗ end_sexpr)()
- void(∗ characters)(const char ∗data, size_t len, atom_t aty)
- void(∗ binary)(const char ∗data, size_t len)

7.3.1 Detailed Description

Some users would prefer to, instead of parsing a full string and walking a potentially huge sexp_t structure, use an XML SAX-style parser where events are triggered as certain parts of the s-expression are encountered. This structure contains a set of function pointers that are called by the parser as it hits expression start and end, and completes reading atoms and binary data. NOTE: The parser_event_handler struct that is a field in the continuation data structure is NOT freed by destroy_continuation since structs for callbacks are ALWAYS malloc'd by the user, not the library.

7.3.2 Field Documentation

7.3.2.1 void(∗ binary)(const char ∗data, size_t len)

The binary function pointer is called when the parser is functioning in INLINE_BINARY mode and binary data is encountered. The function must take two arguments: a pointer to the beginning of the binary data and the number of bytes of data present.

7.3.2.2 void(∗ characters)(const char ∗data, size_t len, atom_t aty)

The characters function pointer is called when an atom is completely parsed. The function must take three arguments: a pointer to the atom data, the number of elements the atom contains, and the specific type of atom that the data represents.

7.3.2.3 void(∗ end_sexpr)()

The end_sexpr function pointer is called when an close parenthesis is encountered ending an expression.

7.3.2.4 void(∗ start_sexpr)()

The start_sexpr function pointer is called when an open parenthesis is encountered starting an expression.

7.4 pcont Struct Reference

`#include <sexp.h>`

Data Fields

- faststack_t * stack
- sexp_t * last_sexp
- char * val
- size_t val_allocated
- size_t val_used
- char * vcur
- char * lastPos
- char * sbuffer
- unsigned int depth
- unsigned int qdepth
- unsigned int state
- unsigned int esc
- unsigned int squoted
- sexp_errcode_t error
- parsermode_t mode
- size_t binexpected
- size_t binread
- char * bindata
- parser_event_handlers_t * event_handlers

7.4.1 Detailed Description

A continuation is used by the parser to save and restore state between invocations to support partial parsing of strings. For example, if we pass the string "(foo bar)(goo car)" to the parser, we want to be able to retrieve each s-expression one at a time - it would be difficult to return all s-expressions at once without knowing how many there are in advance (this would require more memory management than we want...). So, by using a continuation-based parser, we can call it with this string and have it return a continuation when it has parsed the first s-expression. Once we have processed the s-expression (accessible through the *last_sexpr* field of the continuation), we can call the parser again with the same string and continuation, and it will be able to pick up where it left off.

We use continuations instead of a state-ful parser to allow multiple concurrent strings to be parsed by simply maintaining a set of continuations. Manipulating continuations by hand is required if the continuation-based parser is called directly. This is **not recommended** unless you are willing to deal with potential errors and are willing to

learn exactly how the continuation relates to the internals of the parser. A simpler approach is to use either the *parse_sexp* function that simply returns an s-expression without exposing the continuations, or the *iparse_sexp* function that allows iteratively popping one s-expression at a time from a string containing one or more s-expressions. Refer to the documentation for each parsing function for further details on behavior and usage.

7.4.2 Field Documentation

7.4.2.1 char* bindata

Pointer to the memory containing the binary data being read in.

7.4.2.2 size_t binexpected

Length to expect of the current binary data being read in. this also corresponds to the size of the memory allocated for reading this binary data into.

7.4.2.3 size_t binread

Number of bytes of the binary blob that have already been read in.

7.4.2.4 unsigned int depth

This is the depth of parenthesis (the number of left parens encountered) that the parser is currently working with.

7.4.2.5 sexp_errcode_t error

Error code. Used to indicate that the continuation being returned does not represent a successful parsing and thus the contents aren't of much value.

7.4.2.6 unsigned int esc

This is a flag indicating whether the next character to be processed should be assumed to have been prefaced with a '\' character to escape it.

7.4.2.7 parser_event_handlers_t* event_handlers

Pointer to a structure holding handlers for sexpr events. NULL for normal parser operation. This field is NOT freed by destroy_continuation and must be free'd by the user.

This is because these are malloc'd outside the library ALWAYS, so they are the user's responsibility.

7.4.2.8 sexp_t* last_sexp

The last full s-expression encountered by the parser. If this is NULL, the parser has not encountered a full s-expression and more data is required for the current s-expression being parsed. If this is non-NULL, then the parser has encountered one s-expression and may be partially through parsing the next s-expression.

7.4.2.9 char* lastPos

Pointer to the last character to examine in the string being parsed. When the parser is called with the continuation, this is the first character that will be processed. If this is NULL, the parser will start parsing at the beginning of the string passed into the parser.

7.4.2.10 parsermode_t mode

Mode. The parsers' specialized behaviours can be activated by tweaking the mode setting. There are currently two available: normal and inline_binary. Inline_binary treats atoms that start with #b# specially, assuming that they have the structure:

#b#s#data

Where s is a positive (greater than 0) integer representing the length of the data, and data is s bytes of binary data following the # sign. After the s bytes, it is assumed normal s-expression data continues.

7.4.2.11 unsigned int qdepth

This is the depth of parenthesis encountered after a single quote (tick) if the character immediately following the tick was a left paren.

7.4.2.12 char* sbuffer

This is a pointer to the beginning of the current string being processed. lastPos is a pointer to some value inside the string that this points to.

7.4.2.13 unsigned int squoted

Flag whether or not we are processing an atom that was preceeded by a single quote.

7.4.2.14 faststack_t* stack

The parser stack used for iterative parsing.

7.4.2.15 unsigned int state

This is the state ID of the current state of the parser in the DFA representing the parser. The current parser is a DFA based parser to simplify restoring the proper state from a continuation.

7.4.2.16 char* val

Pointer to a temporary buffer used to store atom values during parsing.

7.4.2.17 size_t val_allocated

Current number of bytes allocated for val.

7.4.2.18 size_t val_used

Current number of used bytes in val.

7.4.2.19 char* vcur

Pointer to the character following the last character in the current atom value being parsed.

7.5 sexp_iowrap Struct Reference

`#include <sexp.h>`

Data Fields

- pcont_t * cc
- int fd
- char buf [BUFSIZ]
- int cnt

7.5.1 Detailed Description

This structure is a wrapper around a standard I/O file descriptor and the parsing infrastructure (continuation and a buffer) required to parse off of it. This is used so that routines can hide the loops and details required to accumulate up data read off of the file descriptor and parse expressions individually out of it.

7.5.2 Field Documentation

7.5.2.1 char buf[BUFSIZ]

Buffer to read data into before parsing.

7.5.2.2 pcont_t* cc

Continuation used to parse off of the file descriptor.

7.5.2.3 int cnt

Byte count for last read. If it is -1, there was an error. Otherwise, it will be a value from 0 to BUFSIZ.

7.5.2.4 int fd

The file descriptor. Currently CANNOT be a socket since implementation uses read(), not recv().

7.6 stack_level Struct Reference

`#include <faststack.h>`

Data Fields

- stack_level * above
- stack_level * below
- void * data

7.6.1 Detailed Description

Structure representing a single level in the stack. Has a pointer to the level above and below itself and a pointer to a generic blob of data associated with this level.

7.6.2 Field Documentation

7.6.2.1 struct stack_level* above

Pointer to the level above this one. If NULL, then this level is the top of the stack. If above is non-NULL, this level *may* be the top, but all that can be guaranteed is that there are other allocated but potentially unused levels above this one.

7.6.2.2 struct stack_level* below

Pointer to the level below this one. If NULL, then this level is the bottom.

7.6.2.3 void* data

Pointer to some data associated with this level. User is responsible for knowing what to cast the `void *` pointer into.

7.7 stack_wrapper Struct Reference

`#include <faststack.h>`

Data Fields

- stack_lvl_t * top
- stack_lvl_t * bottom
- int height

7.7.1 Detailed Description

Wrapper around the stack levels - keeps a pointer to the current top and bottom of the stack and a count of the current height. This allows the top to have non-null above pointer resulting from previously allocated stack levels that may be recycled later without `malloc` overhead.

7.7.2 Field Documentation

7.7.2.1 stack_lvl_t* bottom

The bottom of the stack. If this is NULL, the stack is empty.

7.7.2.2 int height

The current height of the stack, in terms of allocated and used levels.

7.7.2.3 stack_lvl_t* top

The top of the stack. If this is NULL, the stack is empty.

Chapter 8

Small, Fast S-Expression Library File Documentation

8.1 cstring.h File Reference

```
#include <stdlib.h>
```

Data Structures

- struct __cstring

Typedefs

- typedef __cstring CSTRING

Functions

- void sgrowsize (size_t s)
- CSTRING * snew (size_t s)
- CSTRING * sadd (CSTRING *s, char *a)
- CSTRING * saddch (CSTRING *s, char a)
- CSTRING * strim (CSTRING *s)
- char * toCharPtr (CSTRING *s)
- void sempty (CSTRING *s)
- void sdestroy (CSTRING *s)

8.1.1 Typedef Documentation

8.1.1.1 typedef struct __cstring CSTRING

Structure wrapping the character pointer and size counters (allocated vs. actual used).

8.1.2 Function Documentation

8.1.2.1 CSTRING* sadd (CSTRING * *s*, char * *a*)

Concatenate the second argument to the CSTRING passed in the first. The second argument must be a pointer to a null terminated string. A NULL return value indicates that something went wrong and that sexp_errno should be checked for the cause. The contents of s are left alone. As such, the caller should check the pointer returned before overwriting the value of s, as this may result in a memory leak if an error condition occurs.

8.1.2.2 CSTRING* saddch (CSTRING * *s*, char *a*)

Append a character to the end of the CSTRING. A NULL return value indicates that something went wrong and that sexp_errno should be checked for the cause. The contents of s are left alone. As such, the caller should check the pointer returned before overwriting the value of s, as this may result in a memory leak if an error condition occurs.

8.1.2.3 void sdestroy (CSTRING * *s*)

Destroy the CSTRING struct and the data it points at.

8.1.2.4 void sempty (CSTRING * *s*)

Set the current length to zero, effectively dumping the string without deallocating it so we can use it later without reallocating any memory.

8.1.2.5 void sgrowsize (size_t *s*)

Set the growth size. Values less than one are ignored.

8.1.2.6 CSTRING* snew (size_t *s*)

Allocate a new CSTRING of the given size. A NULL return value indicates that something went wrong and that sexp_errno should be checked for the cause.

8.1.2.7 CSTRING* strim (CSTRING * *s*)

Trim the allocated memory to precisely the string length plus one char to hold the null terminator A NULL return value indicates that something went wrong and that sexp_errno should be checked for the cause. The contents of s are left alone. As such, the caller should check the pointer returned before overwriting the value of s, as this may result in a memory leak if an error condition occurs.

8.1.2.8 char* toCharPtr (CSTRING * *s*)

Return the base pointer of the CSTRING. NULL either means the base pointer was null, or the CSTRING itself was NULL.

8.2 faststack.h File Reference

Implementation of a fast stack with smart memory management.

Data Structures

- struct stack_level
- struct stack_wrapper

Defines

- #define top_data(s) (s → top → data)
- #define empty_stack(s) (s → top == NULL)

Typedefs

- typedef stack_level stack_lvl_t
- typedef stack_wrapper faststack_t

Functions

- faststack_t * make_stack ()
- void destroy_stack (faststack_t *s)
- faststack_t * push (faststack_t *cur_stack, void *data)
- stack_lvl_t * pop (faststack_t *s)

8.2.1 Detailed Description

Implementation of a fast stack with smart memory management.

8.2.2 Define Documentation

8.2.2.1 #define empty_stack(s) (s → top == NULL)

Given a stack *s*, check to see if the stack is empty or not. Value is boolean true or false.

8.2.2.2 #define top_data(s) (s → top → data)

Given a stack *s*, examine the data pointer at the top.

8.2.3 Typedef Documentation

8.2.3.1 typedef struct stack_wrapper faststack_t

Wrapper around the stack levels - keeps a pointer to the current top and bottom of the stack and a count of the current height. This allows the top to have non-null above pointer resulting from previously allocated stack levels that may be recycled later without `malloc` overhead.

8.2.3.2 typedef struct stack_level stack_lvl_t

Structure representing a single level in the stack. Has a pointer to the level above and below itself and a pointer to a generic blob of data associated with this level.

8.2.4 Function Documentation

8.2.4.1 void destroy_stack (faststack_t * *s*)

Given a stack structure, destroy it and free all of the stack levels. **Important note** : This function *does not* free the data pointed to from each level of the stack - the user is responsible for freeing this data themselves before calling this function to prevent memory leakage.

8.2.4.2 faststack_t* make_stack ()

Return a pointer to an empty stack structure. If the return value is NULL, one should check sexp_errno to determine why.

8.2.4.3 stack_lvl_t* pop (faststack_t * *s*)

Given a stack, pop a level off and return a pointer to that level. The user is responsible for extracting the data, but the stack_lvl_t structures pointed to from the level (above and below) should be left alone. If NULL is returned, either the stack contained nothing, or the incoming stack s was NULL. Consult sexp_errno to determine which was the case – SEXP_ERR_BAD_STACK indicates a null stack was passed in.

8.2.4.4 faststack_t* push (faststack_t * *cur_stack*, void * *data*)

Given a stack, push a new level on referring to the data pointer. If a new level cannot be allocated, NULL is returned and sexp_errno is set with the appropriate error condition. Memory allocation errors will result in SEXP_ERR_MEMORY, while a null stack will result in SEXP_ERR_BAD_STACK.

8.3 sexp.h File Reference

API for a small, fast and portable s-expression parser library.

```
#include <stddef.h>
#include <stdio.h>
#include "faststack.h"
#include "cstring.h"
#include "sexp_memory.h"
#include "sexp_errors.h"
#include "sexp_ops.h"
```

Data Structures

- struct elt
- struct parser_event_handlers
- struct pcont
- struct sexp_iowrap

Typedefs

- typedef elt sexp_t
- typedef parser_event_handlers parser_event_handlers_t
- typedef pcont pcont_t
- typedef sexp_iowrap sexp_iowrap_t

Enumerations

- enum elt_t { SEXP_VALUE, SEXP_LIST }
- enum atom_t { SEXP_BASIC, SEXP_SQUOTE, SEXP_DQUOTE, SEXP_BINARY }
- enum parsermode_t { PARSER_NORMAL, PARSER_INLINE_BINARY, PARSER_EVENTS_ONLY }

Functions

- sexp_errcode_t set_parser_buffer_params (size_t ss, size_t gs)
- sexp_t * sexp_t_allocate (void)
- void sexp_t_deallocate (sexp_t *s)

- void sexp_cleanup (void)
- int print_sexp (char *loc, size_t size, const sexp_t *e)
- int print_sexp_cstr (CSTRING **s, const sexp_t *e, size_t ss)
- sexp_t * new_sexp_list (sexp_t *l)
- sexp_t * new_sexp_atom (const char *buf, size_t bs, atom_t aty)
- pcont_t * init_continuation (char *str)
- void destroy_continuation (pcont_t *pc)
- sexp_iowrap_t * init_iowrap (int fd)
- void destroy_iowrap (sexp_iowrap_t *iow)
- sexp_t * read_one_sexp (sexp_iowrap_t *iow)
- sexp_t * parse_sexp (char *s, size_t len)
- sexp_t * iparse_sexp (char *s, size_t len, pcont_t *cc)
- pcont_t * cparse_sexp (char *s, size_t len, pcont_t *pc)
- void destroy_sexp (sexp_t *s)
- void reset_sexp_errno ()

Variables

- sexp_errcode_t sexp_errno

8.3.1 Detailed Description

API for a small, fast and portable s-expression parser library.

8.3.2 Typedef Documentation

8.3.2.1 typedef struct parser_event_handlers parser_event_handlers_t

Some users would prefer to, instead of parsing a full string and walking a potentially huge sexp_t structure, use an XML SAX-style parser where events are triggered as certain parts of the s-expression are encountered. This structure contains a set of function pointers that are called by the parser as it hits expression start and end, and completes reading atoms and binary data. NOTE: The parser_event_handler struct that is a field in the continuation data structure is NOT freed by destroy_continuation since structs for callbacks are ALWAYS malloc'd by the user, not the library.

8.3.2.2 typedef struct pcont pcont_t

A continuation is used by the parser to save and restore state between invocations to support partial parsing of strings. For example, if we pass the string "(foo bar)(goo car)" to the parser, we want to be able to retrieve each s-expression one at a time - it

would be difficult to return all s-expressions at once without knowing how many there are in advance (this would require more memory management than we want...). So, by using a continuation-based parser, we can call it with this string and have it return a continuation when it has parsed the first s-expression. Once we have processed the s-expression (accessible through the *last_sexpr* field of the continuation), we can call the parser again with the same string and continuation, and it will be able to pick up where it left off.

We use continuations instead of a state-ful parser to allow multiple concurrent strings to be parsed by simply maintaining a set of continuations. Manipulating continuations by hand is required if the continuation-based parser is called directly. This is **not recommended** unless you are willing to deal with potential errors and are willing to learn exactly how the continuation relates to the internals of the parser. A simpler approach is to use either the *parse_sexp* function that simply returns an s-expression without exposing the continuations, or the *iparse_sexp* function that allows iteratively popping one s-expression at a time from a string containing one or more s-expressions. Refer to the documentation for each parsing function for further details on behavior and usage.

8.3.2.3 typedef struct elt sexp_t

An s-expression is represented as a linked structure of elements, where each element is either an *atom* or *list*. An atom corresponds to a string, while a list corresponds to an s-expression. The following grammar represents our definition of an s-expression:

```
sexpr  ::= ( sx )
sx     ::= atom sxtail | sexpr sxtail | 'sexpr sxtail | 'atom sxtail | NULL
sxtail ::= sx | NULL
atom   ::= quoted | value
quoted ::= "ws_string"
value  ::= nws_string
```

An atom can either be a quoted string, which is a string containing whitespace (possibly) surrounded by double quotes, or a non-whitespace string that does not require surrounding quotes. An element representing an atom will have a type of *value* and data stored in the *val* field. An element of type *list* represents an s-expression corresponding to *sexpr* in the grammar, and will have a pointer to the head of the appropriate s-expression. Details regarding these fields and their values given with the fields themselves. Notice that a single quote can appear directly before an s-expression or atom, similar to the use in LISP.

8.3.3 Enumeration Type Documentation

8.3.3.1 enum atom_t

For an element that represents a value, the value can be interpreted as a more specific type. A *basic* value is a simple string with no whitespace (and therefore no quotes required). A *double quote* value, or *dquote*, is one that contains characters (such as whitespace) that requires quotation marks to contain the string. A *single quote* value, or *squote*, represents an element that is prefaced with a single tick-mark. This can be either an atom or s-expression, and the result is that the parser does not attempt to parse the element following the tick mark. It is simply stored as text. This is similar to the meaning of a tick mark in the Scheme or LISP family of programming languages. Finally, *binary* allows raw binary to be stored within an atom. Note that if the binary type is used, the data is stored in bindata with the length in binlength. Otherwise, the data us stored in the val field with val_used and val_allocated tracking the size of the value string and the total memory allocated for it.

Enumerator:

SEXP_BASIC Basic, unquoted value.

SEXP_SQUOTE Single quote (tick-mark) value - contains a string representing a non-parsed portion of the s-expression.

SEXP_DQUOTE Double-quoted string. Similar to a basic value, but potentially containing white-space.

SEXP_BINARY Binary data. This is used when the specialized parser is active and supports inlining of binary blobs of data inside an expression.

8.3.3.2 enum elt_t

An element in an s-expression can be one of three types: a *value* represents an atom with an associated text value. A *list* represents an s-expression, and the element contains a pointer to the head element of the associated s-expression.

Enumerator:

SEXP_VALUE An atom of some type. See atom type (aty) field of element structure for details as to which atom type this is.

SEXP_LIST A list. This means the element points to an element representing the head of a list.

8.3.3.3 enum parsermode_t

parser mode flag used by continuation to toggle special parser behaviour.

Enumerator:

PARSER_NORMAL normal (LISP-style) s-expression parser behaviour.

PARSER_INLINE_BINARY treat atoms beginning with #b# as inlined binary data. everything else is treated the same as in PARSER_NORMAL mode.

PARSER_EVENTS_ONLY if the event_handlers field in the continuation contains a non-null value, the handlers specified in the parser_event_handlers_t struct will be called as appropriate, but the parser will not allocate a structure composed of sexp_t structs. Note that if the event_handlers is set to null and this mode is selected, the user would be better off not calling anything in the first place, as they are telling the parser to walk the string, but do nothing productive in the process.

8.3.4 Function Documentation

8.3.4.1 void destroy_continuation (pcont_t * *pc*)

destroy a continuation. This involves cleaning up what it contains, and cleaning up the continuation itself.

8.3.4.2 void destroy_sexp (sexp_t * *s*)

given a sexp_t structure, free the memory it uses (and recursively free the memory used by all sexp_t structures that it references). Note that this will call the deallocation routine for sexp_t elements. This means that memory isn't freed, but stored away in a cache of pre-allocated elements. This is an optimization to speed up the parser to eliminate wasteful free and re-malloc calls. Note: If using inlined binary mode, this will free the data pointed to by the bindata field. So, if you care about the data after the lifetime of the s-expression, make sure to make a copy before cleaning up the sexpr.

8.3.4.3 pcont_t* init_continuation (char * *str*)

create an initial continuation for parsing the given string

8.3.4.4 sexp_t* new_sexp_atom (const char * *buf*, size_t *bs*, atom_t *aty*)

Allocate a new sexp_t element representing a value. The user must specify the precise type of the atom. This used to default to SEXP_BASIC, but this can lead to errors if the user did not expect this assumption. By explicitly passing in the atom type, the caller should ensure that the data in the buffer is valid given the requested atom type. For performance reasons, such checks are left to the caller if they are desired, and not performed in the library if they are not wanted.

8.3.4.5 sexp_t* new_sexp_list (sexp_t * *l*)

Allocate a new sexp_t element representing a list.

8.3.4.6 int print_sexp (char * *loc*, size_t *size*, const sexp_t * *e*)

print a sexp_t struct as a string in the LISP style. If the buffer is large enough and the conversion is successful, the return value represents the length of the string contained in the buffer. If the buffer was too small, or some other error occurred, the return value is -1 and the contents of the buffer should not be assumed to contain any useful information. When the return value is -1, the caller should check the contents of sexp_errno for details on what error may have occurred.

8.3.4.7 int print_sexp_cstr (CSTRING ** *s*, const sexp_t * *e*, size_t *ss*)

print a sexp_t structure to a buffer, growing it as necessary instead of relying on fixed size buffers like print_sexp. Important argument to tune for performance reasons is `ss` - the buffer start size. The growsize used by the CSTRING routines also should be considered for tuning via the sgrowsize() function. This routine no longer requires the user to specify the growsize, and uses the current setting without changing it.

8.3.4.8 void reset_sexp_errno ()

reset the value of sexp_errno to SEXP_ERR_OK.

8.3.4.9 void sexp_cleanup (void)

In the event that someone wants us to release ALL of the memory used between calls by the library, they can free it. If you don't call this, the caches will be persistent for the lifetime of the library user. Note that in the event of an error condition resulting in sexp_errno being set, the user might consider calling this to clean up any memory that may be lingering around that should be cleaned up.

8.3.4.10 sexp_t* sexp_t_allocate (void)

return an allocated sexp_t. This structure may be an already allocated one from the stack or a new one if none are available. Use this instead of manually mallocing if you want to avoid excessive mallocs. *Note: Mallocing your own expressions is fine - you can even use sexp_t_deallocate to deallocate them and put them in the pool.* Also, if the stack has not been initialized yet, this does so.

8.3.4.11 void sexp_t_deallocate (sexp_t * *s*)

given a malloc'd sexp_t element, put it back into the already-allocated element stack. This method will allocate a stack if one has not been allocated already.

8.3.5 Variable Documentation

8.3.5.1 sexp_errcode_t sexp_errno

Global value indicating the most recent error condition encountered. This value can be reset to SEXP_ERR_OK by calling sexp_errno_reset().

8.4 sexp_errors.h File Reference

Error conditions are enumerated here along with any routines for translating error codes to human readable messages.

Enumerations

- enum sexp_errcode_t {
  SEXP_ERR_OK = 0, SEXP_ERR_MEMORY, SEXP_ERR_BADFORM, SEXP_ERR_BADCONTENT,
  SEXP_ERR_NULLSTRING, SEXP_ERR_IO, SEXP_ERR_IO_EMPTY, SEXP_ERR_MEM_LIMIT,
  SEXP_ERR_BUFFER_FULL, SEXP_ERR_BAD_PARAM, SEXP_ERR_BAD_STACK, SEXP_ERR_UNKNOWN_STATE }

8.4.1 Detailed Description

Error conditions are enumerated here along with any routines for translating error codes to human readable messages.

8.4.2 Enumeration Type Documentation

8.4.2.1 enum sexp_errcode_t

Error codes used by the library are defined in this enumeration. They are either used as values for the error field within the continuation structures, or as return values for functions with a return type of sexp_errcode_t.

Enumerator:

SEXP_ERR_OK no error.

SEXP_ERR_MEMORY memory error. malloc/realloc/calloc failures may result in this error code. this can either result from the system calls failing, or in the limited memory mode of the library, the memory limit being exceeded. In limited memory mode, if this error condition is present, one should check the memory limits that were in place during the erroneous call.

SEXP_ERR_BADFORM badly formed expression. Missing, misplaced, or mismatched parenthesis will result in this error.

SEXP_ERR_BADCONTENT a sexp_t that is inconsistent will result in this error code. An example is a SEXP_BASIC sexp_t with a null val field but a non-zero val_used value. Similar cases exist for SEXP_DQUOTE, SQUOTE, and BINARY types. This value is also used in the parser to flag a case where an inlined binary block is given a negative length.

SEXP_ERR_NULLSTRING if a null string is passed into the parser, this error occurs.

SEXP_ERR_IO general IO related errors, such as failure of fopen(). these are basically non-starters with respect to getting the IO routines going.

SEXP_ERR_IO_EMPTY I/O routines that return NULL may simply have nothing to read. This is sometimes an error condition if the io wrapper continuation contains a partially complete s-expression, but nothing more is present (yet) on the file descriptor.

SEXP_ERR_MEM_LIMIT when running the library under limited memory (ie, _SEXP_LIMIT_MEMORY_ defined at build time), this error will be produced when the memory limit is exceeded.

SEXP_ERR_BUFFER_FULL buffer for unparsing is full.

SEXP_ERR_BAD_PARAM routines that take parameters such as memory limits, growth sizes, or default sizes, can produce this error if a bad value has been passed in. this error usually will indicate that the parameters were bad and the default values were used instead (ie, it is non-fatal.).

SEXP_ERR_BAD_STACK bad stack state encountered.

SEXP_ERR_UNKNOWN_STATE unknown parser state

8.5 sexp_memory.h File Reference

Wrappers around basic memory allocation/deallocation routines to allow memory usage limiting. Only enabled if _SEXP_LIMIT_MEMORY_ is defined when building the library, otherwise the routines are defined to be the standard malloc/calloc/realloc/free functions.

Defines

- #define sexp_calloc(count, size) calloc(count,size)
- #define sexp_malloc(size) malloc(size)
- #define sexp_free(ptr, size) free(ptr)
- #define sexp_realloc(ptr, size, oldsize) realloc((ptr),(size))

8.5.1 Detailed Description

Wrappers around basic memory allocation/deallocation routines to allow memory usage limiting. Only enabled if _SEXP_LIMIT_MEMORY_ is defined when building the library, otherwise the routines are defined to be the standard malloc/calloc/realloc/free functions.

8.5.2 Define Documentation

8.5.2.1 #define sexp_calloc(count, size) calloc(count,size)

_SEXP_LIMIT_MEMORY_ not defined. This is a macro that maps to calloc().

8.5.2.2 #define sexp_free(ptr, size) free(ptr)

_SEXP_LIMIT_MEMORY_ not defined. This is a macro that maps to free().

8.5.2.3 #define sexp_malloc(size) malloc(size)

_SEXP_LIMIT_MEMORY_ not defined. This is a macro that maps to malloc().

8.5.2.4 #define sexp_realloc(ptr, size, oldsize) realloc((ptr),(size))

_SEXP_LIMIT_MEMORY_ not defined. This is a macro that maps to realloc().

8.6 sexp_ops.h File Reference

A collection of useful operations to perform on s-expressions.

`#include "sexp.h"`

Defines

- #define hd_sexp(s) ((s) → list)
- #define tl_sexp(s) ((s) → list → next)
- #define next_sexp(s) ((s) → next)
- #define reset_pcont(c) ((c) → lastPos = NULL)

Functions

- sexp_t * find_sexp (const char *name, sexp_t *start)
- sexp_t * bfs_find_sexp (const char *name, sexp_t *start)
- int sexp_list_length (const sexp_t *sx)
- sexp_t * copy_sexp (const sexp_t *sx)

8.6.1 Detailed Description

A collection of useful operations to perform on s-expressions.

A set of routines for operating on s-expressions.

8.6.2 Define Documentation

8.6.2.1 #define hd_sexp(s) ((s) → list)

Return the head of a list *s* by reference, not copy.

8.6.2.2 #define next_sexp(s) ((s) → next)

Return the element following the argument *s*.

8.6.2.3 #define reset_pcont(c) ((c) → lastPos = NULL)

Reset the continuation *c* by setting the `lastPos` pointer to `NULL`.

8.6.2.4 #define tl_sexp(s) ((s) → list → next)

Return the tail of a list *s* by reference, not copy.

8.6.3 Function Documentation

8.6.3.1 sexp_t* bfs_find_sexp (const char * *name*, sexp_t * *start*)

Breadth first search for s-expressions. Depth first search will find the first occurance of a string in an s-expression by basically finding the earliest occurance in the string representation of the expression itself. Breadth first search will find the first occurance of a string in relation to the structure of the expression itself (IE: the instance with the lowest depth will be found).

Parameters:

name Value to search for.

start Root element of the s-expression to search from.

Returns:

If the value is found, return a pointer to the first occurrence in a breadth-first traversal. NULL if not found.

8.6.3.2 sexp_t* copy_sexp (const sexp_t * *sx*)

Copy an s-expression. This is a deep copy - so the resulting s-expression shares no pointers with the original. The new one can be changed without damaging the contents of the original.

Parameters:

sx S-expression to copy.

Returns:

A pointer to a copy of sx. This is a deep copy, so no memory is shared between the original and the returned copy.

8.6.3.3 sexp_t* find_sexp (const char * *name*, sexp_t * *start*)

Find an atom in a sexpression data structure and return a pointer to it. Return NULL if the string doesn't occur anywhere as an atom. This is a depth-first search algorithm.

Parameters:

name Value to search for.

start Root element of the s-expression to search from.

Returns:
If the value is found, return a pointer to the first occurrence in a depth-first traversal. NULL if not found.

8.6.3.4 int sexp_list_length (const sexp_t * *sx*)

Given an s-expression, determine the length of the list that it encodes. A null expression has length 0. An atom has length 1. A list has length equal to the number of sexp_t elements from the list head to the end of the ->next linked list from that point.

Parameters:
sx S-expression input.

Returns:
Number of sexp_t elements at the same level as sx, 0 for NULL, 1 for an atom.

8.7 sexp_vis.h File Reference

API for emitting graphviz data structure visualizations.

Functions

- sexp_errcode_t sexp_to_dotfile (const sexp_t *sx, const char *fname)

8.7.1 Detailed Description

API for emitting graphviz data structure visualizations.

Index