Stream: Internet Engineering Task Force (IETF)
RFC: 9804
Category: Informational
Published: June 2025
ISSN: 2070-1721
Authors: R. Rivest *MIT CSAIL* D. Eastlake 3rd *Independent*

RFC 9804
Simple Public Key Infrastructure (SPKI) S-Expressions

Abstract

This memo specifies the data structure representation that was devised to support Simple Public Key Infrastructure (SPKI) certificates, as detailed in RFC 2692, with the intent that it be more widely applicable. It has been and is being used elsewhere. There are multiple implementations in a variety of programming languages. Uses of this representation are referred to in this document as "S-expressions". This memo makes precise the encodings of these SPKI S-expressions: It gives a "canonical form" for them, describes two "transport" representations, and also describes an "advanced" format for display to people.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9804.

Copyright Notice

Table of Contents

1. Introduction

This memo specifies the data structure representation that was devised to support Simple Public Key Infrastructure (SPKI) certificates [RFC2692], with the intent that it be more widely applicable (see Section 1.3, "Historical Note"). It is suitable for representing arbitrary, complex data structures and has been and is being used elsewhere. Uses of this representation herein are referred to as "S-expressions".

This memo makes precise the encodings of these SPKI S-expressions: It gives a "canonical form" for them, describes two "transport" representations, and also describes an "advanced" format for display to people. There are multiple implementations of S-expressions in a variety of programming languages including Python, Ruby, and C (see Appendix A).

These S-expressions are either octet-strings or lists of simpler S-expressions. Here is a sample S-expression:

```
    (snicker "abc" (#03# |YWJj|))
```

It is a list of length three containing the following:

- the octet-string "snicker"
- the octet-string "abc"
- a sub-list containing two elements: The hexadecimal constant #03# (which represents a one-octet-long octet-string with the value of that octet being 0x03) and the base-64 constant |YWJj| (which represents the same octet-string as "abc")

This document specifies how to construct and use these S-expressions.

The design goals for S-expressions were as follows:

- Generality: S-expressions should be good at representing arbitrary data.
- Readability: It should be easy for someone to examine and understand the structure of an S-expression.
- Economy: S-expressions should represent data compactly.
- Transportability: S-expressions should be easy to transport over communication media (such as email) that are known to be less than perfect.
- Flexibility: S-expressions should make it relatively simple to modify and extend data structures.
- Canonicalization: It should be easy to produce a unique "canonical" form of an S-expression, for digital signature purposes.
- Efficiency: S-expressions should admit in-memory representations that allow efficient processing.

For implementors of new applications and protocols other technologies also worthy of consideration include the following: XML [XML], CBOR [RFC8949], and JSON [RFC8259].

1.1. Uses of S-Expressions

The S-expressions specified herein are in active use today between GnuPG [GnuPG] and Ribose's RNP [Ribose]. Ribose has implemented C++ software to compose and parse these S-expressions [RNPGP_SEXPP]. The GNU software is the Libgcrypt library [Libgcrypt], and there are other implementations (see Appendix A).

S-expressions are also used or referenced in the following RFCs:

- [RFC2693] for [SPKI]
- [RFC3275] XML-Signature Syntax and Processing

In addition, S-expressions are the inspiration for the encodings in other protocols. For example, [RFC3259] or Section 6 of [CDDL-freezer].

1.2. Formalization

[Formal] is an Internet-Draft that shows a formal model of SPKI S-expressions and formally demonstrates that the examples and ABNF in this document are correct.

1.3. Historical Note

The S-expressions described here were originally developed for "SDSI" (the Simple Distributed Security Infrastructure by Lampson and Rivest [SDSI]) in 1996, although their origins date back to McCarthy's [LISP] programming language. They were further refined and improved during the merger of SDSI and SPKI [SPKI] [RFC2692] [RFC2693] during the first half of 1997. S-expressions are more readable and flexible than Bernstein's "netstrings" [BERN], which were developed contemporaneously.

> Although a specification was made publicly available as a file named draft-rivest-sexp-00.txt on 4 May 1997, that file was never actually submitted to the IETF. This document is a clarified and modernized version of that document.

1.4. Conventions Used in This Document

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. S-expressions -- Informal Introduction

Informally, an S-expression is either:

- an octet-string, or
- a finite list of simpler S-expressions.

An octet-string is a finite sequence of eight-bit octets. An octet-string may be zero length. There may be many different but equivalent ways of representing an octet-string

```
    abc         -- as a token
    "abc"       -- as a quoted string
    #616263#    -- as a hexadecimal string
    3:abc       -- as a length-prefixed "verbatim" encoding
    |YWJj|      -- as a base-64 encoding of the octet-string "abc"
```

The above encodings are all equivalent in that they all denote the same octet-string. Details of these encodings are given below, and how to give a "display type" to a simple-string is also described in Section 4.6.

A list is a finite sequence of zero or more simpler S-expressions. A list is represented by using parentheses to surround the sequence of encodings of its elements, as in:

```
    (abc (de #6667#) "ghi jkl")
```

As can be seen, there is variability possible in the encoding of an S-expression. In some applications, it is desirable to standardize or restrict the encodings; in other cases, it is desirable to have no restrictions. The following are the target cases these S-expressions aim to handle:

- a "transport" or "basic" encoding for transporting the S-expression between computers
- a "canonical" encoding, used when signing the S-expression
- an "advanced" encoding used for input/output to people
- an "in-memory" encoding used for processing the S-expression in the computer

In this document, related encoding techniques for each of these uses are provided.

3. Character Set

This document specifies encodings of S-expressions. Except when giving "verbatim" encodings, the character set used is limited to the following characters in ASCII [RFC0020]:

Alphabetic:

```
    A B ... Z a b ... z
```

Numeric:

```
    0 1 ... 9
```

Whitespace:

```
    space, horizontal tab, vertical tab, form-feed
    carriage-return, line-feed
```

The following graphics characters, which are called "pseudo-alphabetic" in this document:

```
    -  hyphen or minus
    .  period
    /  slash
    _  underscore
    :  colon
    *  asterisk
    +  plus
    =  equal
```

The following graphics characters, which are "reserved punctuation":

```
     (  left parenthesis
     )  right parenthesis
     [  left bracket
     ]  right bracket
     {  left brace
     }  right brace
     |  vertical bar
     #  number sign
     "  double quote
     &  ampersand
     \  backslash
```

The following characters are unused and unavailable, except in "verbatim" and "quoted string" encodings:

```
     !  exclamation point
     %  percent
     ^  circumflex
     ~  tilde
     ;  semicolon
     '  single-quote (apostrophe)
     ,  comma
     <  less than
     >  greater than
     ?  question mark
```

4. Octet-String Representation Types

This section describes in detail the ways in which an octet-string may be represented.

Recall that an octet-string is any finite sequence of octets and that an octet-string may have length zero.

4.1. Verbatim Representation

A verbatim encoding of an octet-string consists of three parts:

- the length (number of octets) of the octet-string, given in decimal, most significant digit first, with no leading zeros
- a colon ":"
- the octet-string itself, verbatim

There are no blanks or whitespace separating the parts. No "escape sequences" are interpreted in the octet-string. This encoding is also called a "binary" or "raw" encoding.

Here are some sample verbatim encodings:

```
      3:abc
      7:subject
      4:::":
      12:hello world!
      10:abcdefghij
      0:
```

4.2. Quoted-String Representation

The quoted-string representation of an octet-string consists of:

- an optional decimal length field
- an initial double-quote (")
- the octet-string with the C programming language [C88] escape conventions (\n, etc.)
- a final double-quote (")

The specified length is the length of the resulting string after any backslash escape sequences have been converted to the octet value they denote. The string does not have any "terminating NULL" that [C88] includes, and the length does not count such an octet.

The length is optional.

The escape conventions within the quoted string are as follows (these follow the C programming language [C88] conventions, with an extension for ignoring line terminators of just CR, LF, CRLF, or LFCR and more restrictive octal and hexadecimal value formats):

```
      \a     -- audible alert (bell)
      \b     -- backspace
      \t     -- horizontal tab
      \v     -- vertical tab
      \n     -- new-line
      \f     -- form-feed
      \r     -- carriage-return
      \"     -- double-quote
      \'     -- single-quote
      \?     -- question mark
      \\     -- back-slash
      \ooo   -- character with octal value ooo (all three
                digits MUST be present)
      \xhh   -- character with hexadecimal value hh (both
                digits MUST be present)
      \<carriage-return>   -- causes carriage-return to be ignored.
      \<line-feed>         -- causes line-feed to be ignored.
      \<carriage-return><line-feed>   -- causes
                CRLF to be ignored.
      \<line-feed><carriage-return>   -- causes
                LFCR to be ignored.
```

Here are some examples of quoted-string encodings:

```
"subject"
"hi there"
7"subject"
"\xFE is the same octet as \376"
3"\n\n\n"
"This has\n two lines."
"This has \
 one line."
""
```

4.3. Token Representation

An octet-string that meets the following conditions may be given directly as a "token":

- it does not begin with a digit;
- it contains only characters that are: alphabetic (upper or lower case), numeric, or one of the following eight "pseudo-alphabetic" punctuation marks:

```
-  .  /  _  :  *  +  =
```

- it is length 1 or greater.

Note: Upper and lower case are not equivalent. A token may begin with punctuation, including ":".

Here are some examples of token representations:

```
subject
not-before
:=..
class-of-1997
//example.net/names/smith
*
```

4.4. Hexadecimal Representation

An octet-string may be represented with a hexadecimal encoding consisting of:

- an (optional) decimal length of the octet-string
- a sharp-sign "#"
- a hexadecimal encoding of the octet-string, with each octet represented with two hexadecimal digits, most significant digit first. There **MUST** be an even number of such digits.
- a final sharp-sign "#"

There may be whitespace inserted in the midst of the hexadecimal encoding arbitrarily; it is ignored. It is an error to have characters other than whitespace and hexadecimal digits.

Here are some examples of hexadecimal encodings:

```
   #616263#    -- represents "abc"
   3#616263#   -- also represents "abc"
   # 616
     263 #     -- also represents "abc"
   ##          -- represents the zero-length string
```

4.5. Base-64 Representation of Octet-Strings

An octet-string may be represented in a base-64 encoding [RFC4648] consisting of:

- an (optional) decimal length of the octet-string
- a vertical bar "|"
- the base-64 [RFC4648] encoding of the octet-string.
- a final vertical bar "|"

Base-64 encoding produces four characters of output for each three octets of input. When the length of the input is divided by three:

- if the remainder is one, it produces an output block of length four ending in two equals signs.
- if the remainder is two, it produces an output block of length four ending in one equals sign.

These equals signs **MUST** be included on output, but input routines **MAY** accept inputs where one or two equals signs are dropped.

Whitespace inserted in the midst of the base-64 encoding is ignored. It is an error to have characters other than whitespace and base-64 characters.

Here are some examples of base-64 encodings:

```
   |YWJj|       -- represents "abc"
   | Y W
     J j |      -- also represents "abc"
   3|YWJj|      -- also represents "abc"
   |YWJjZA==|   -- represents "abcd"
   |YWJjZA|     -- also represents "abcd"
   ||           -- represents the zero-length string
```

Note the difference between this base-64 encoding of an octet-string using vertical bars ("| |") and the base-64 encoding of an S-expression using curly braces ("{ }") in Section 6.1.

4.6. Display-Hints and Internationalization

An octet-string can contain any type of data representable by a finite octet-string, e.g., text, a fixed or variable-length integer, or an image. Normally, the application producing and/or consuming S-expressions will understand their structure, the data type, and the encoding of the octet-strings within the S-expressions used by that application. If the octet-string consists of text, use of UTF-8 encoding is **RECOMMENDED** [RFC2130] [RFC3629].

The purpose of a display-hint is to provide information on how to display an octet-string to a user. It has no other function. Many of the media types [RFC2046] work here.

A display-hint is an octet-string representation surrounded by square brackets. There may be whitespace separating the display hint octet-string from the surrounding brackets. Any of the legal octet-string representations may be used for the display-hint string, but a display-hint may not be applied to a display-hint string -- that is, display-hints may not be nested.

A display-hint that can be used for UTF-8-encoded text is shown in the following example where the octet-string represents "böb☺", that is, "bob" with an umlaut over the "o", followed by the Unicode [Unicode] character WHITE SMILING FACE (U+263A).

```
      ["text/plain; charset=utf-8"]"b\xC3\xB7b\xE2\x98\xBA"
```

Every octet-string representation is or is not preceded by a single display-hint. There may be whitespace between the close square bracket and the octet-string to which the hint applies.

Here are some other examples of display-hints:

```
      [image/gif]
      [charset=unicode-1-1]
      [  text/richtext  ]
      ["text/plain; charset=iso-8859-1"]
      [application/postscript]
      [audio/basic]
      ["http://example.com/display-types/funky.html"]
```

An octet-string that has no display-hint may be considered to have a media type [RFC2046] specified by the application or use. In the absence of such a specification, the default is as follows:

```
      [application/octet-stream]
```

When an S-expression is being encoded in one of the representations described in Section 6, any display-hint present is included. If a display-hint is the default, it is not suppressed nor is the default display-hint included in the representation for an octet-string without a display-hint.

4.7. Comparison of Octet-Strings

It is **RECOMMENDED** that two octet-strings be considered equivalent for most computational and algorithmic purposes if and only if they have the same display-hint and the same data octet-strings. However, a particular application might need a different criterion. For example, it might ignore the display hint on comparisons.

Note that octet-strings are "case-sensitive"; the octet-string "abc" is not equal to the octet-string "ABC".

An octet-string without a display-hint may be compared to another octet-string (with or without a display hint) by considering it as an octet-string with the default display-hint specified for the applications or, in the absence of such specification, the general default display-hint specified in Section 4.6 .

5. Lists

Just as with octet-strings, there are variations in representing a list. Whitespace may be used to separate list elements, but they are only required to separate two octet-strings when otherwise the two octet-strings might be interpreted as one, as when one token follows another. To be precise, an octet-string represented as a token (Section 4.3) **MUST** be separated by whitespace from a following token, verbatim representation, or any of the following if they are prefixed with a length: quoted-string, hexadecimal, or base-64 representation. Also, whitespace may follow the initial left parenthesis or precede the final right parenthesis of a list.

Here are some examples of encodings of lists:

```
(a bob c)

( a ( bob c ) ( ( d e ) ( e f ) )  )

(11:certificate(6:issuer3:bob)(7:subject5:alice))

(|ODpFeGFtcGxlIQ==| "1997" murphy 3:XC+)

()
```

6. S-Expression Representation Types

There are three "types" of representation:

- canonical
- basic transport
- advanced transport

The first two **MUST** be supported by any implementation; the last is **OPTIONAL**. As part of basic representation, the base-64 [RFC4648] representation of an S-expression may be used as described in Section 6.1.

6.1. Base-64 Representation of S-Expressions

An S-expression may be represented in a base-64 encoding [RFC4648] consisting of:

- an opening curly brace "{"
- the base-64 [RFC4648] encoding of the S-expression
- a final closing curly brace "}"

Base-64 encoding produces four characters of output for each three octets of input. If the length of the input divided by three leaves a remainder of one or two, it produces an output block of length four ending in two or one equals signs, respectively. These equals signs **MUST** be included on output, but input routines **MAY** accept inputs where one or two equals signs are dropped.

Whitespace inserted in the midst of the base-64 encoding, after the opening curly brace, or before the closing curly brace is ignored. It is an error to have characters other than whitespace and base-64 characters.

Note the difference between this base-64 encoding of an S-expression using curly braces ("{ }") and the base-64 encoding of an octet-string using vertical bars ("| |") in Section 4.5.

6.2. Canonical Representation

This canonical representation is used for digital signature purposes and transport over channels not sensitive to specific octet values. It is uniquely defined for each S-expression. It is not particularly readable, but that is not the point. It is intended to be very easy to parse, reasonably economical, and unique for any S-expression. See [CANON1] and [CANON2].

The "canonical" form of an S-expression represents each octet-string in verbatim mode, and represents each list with no blanks separating elements from each other or from the surrounding parentheses. See also Section 7.2.

Here are some examples of canonical representations of S-expressions:

```
   (6:issuer3:bob)
   (4:icon[12:image/bitmap]9:xxxxxxxxx)
   (7:subject(3:ref5:alice6:mother))
   10:foo)]}>bar
   0:
```

6.3. Basic Transport Representation

There are two forms of the "basic transport" representation:

1. The canonical representation

2. A base-64 [RFC4648] representation of the canonical representation, surrounded by braces (see Section 6.1)

The basic transport representations (see Section 7.3) are intended to provide a universal means of representing S-expressions for transport from one machine to another. The base-64 encoding would be appropriate if the channel over which the S-expression is being sent might be sensitive to octets of some special values, such as an octet of all zero bits (NULL) or an octet of all one bits (DEL), or if the channel is sensitive to "line length" such that occasional line terminating whitespace is needed.

Here are two examples of an S-expression represented in basic transport mode:

```
(1:a1:b1:c)

{KDE6YTE6YjE
   6Yyk= }
```

The second example above is the same S-expression as the first encoded in base-64.

6.4. Advanced Transport Representation

The "advanced transport" representation is intended to provide more flexible and readable notations for documentation, design, debugging, and (in some cases) user interface.

The advanced transport representation allows all of the octet-string representation forms described above in Section 4: quoted strings, base-64, hexadecimal, tokens, representations of strings with omitted lengths, and so on. See Section 7.1.

7. ABNF of the Syntax

ABNF is the Augmented Backus-Naur Form for syntax specifications as defined in [RFC5234]. The ABNF for advanced representation of S-expressions is given first, and the basic and canonical forms are derived therefrom. The rule names below in all capital letters are defined in Appendix B.1 of [RFC5234].

7.1. ABNF for Advanced Transport

```
sexp           =  *whitespace value *whitespace

whitespace     =  SP / HTAB / vtab / CR / LF / ff

vtab           =  %x0B   ; vertical tab

ff             =  %x0C   ; form feed

value          =  string / ("(" *(value / whitespace) ")")

string         =  [display] simple-string
```

```
display        =  "[" *whitespace simple-string *whitespace "]"
                  *whitespace

simple-string  =  verbatim / quoted-string / token / hexadecimal /
                  base-64

verbatim       =  decimal ":" *OCTET
                    ; the length followed by a colon and the exact
                    ; number of OCTETs indicated by the length

decimal        =  %x30 / (%x31-39 *DIGIT)

quoted-string  =  [decimal] DQUOTE *(printable / escaped) DQUOTE

printable      =  %x20-21 / %x23-5B / %x5D-7E
                    ; All US-ASCII printable but double-quote and
                    ; backslash

escaped        =  backslash (%x3F / %x61 / %x62 / %x66 / %x6E /
                  %x72 / %x74 / %x76 / DQUOTE / quote / backslash
                  / 3(%x30-37) / (%x78 2HEXDIG) / CR / LF /
                  (CR LF) / (LF CR))

backslash      =  %x5C

quote          =  %x27   ; single quote

token          =  (ALPHA / simple-punc) *(ALPHA / DIGIT /
                     simple-punc)

simple-punc    =  "-" / "." / "/" / "_" / ":" / "*" / "+" / "="

hexadecimal    =  [decimal] "#" *whitespace *hexadecimals "#"

hexadecimals   =  2(HEXDIG *whitespace)

base-64        =  [decimal] "|" *whitespace *base-64-chars
                     [base-64-end] "|"

base-64-chars  =  4(base-64-char *whitespace)

base-64-char   =  ALPHA / DIGIT / "+" / "/"

base-64-end    =  base-64-chars /
                  3(base-64-char *whitespace) ["=" *whitespace] /
                  2(base-64-char *whitespace) *2("=" *whitespace)
```

7.2. ABNF for Canonical

```
c-sexp         =  c-string / ("(" *c-sexp ")")

c-string       =  [ "[" verbatim "]" ] verbatim
```

7.3. ABNF for Basic Transport

```
b-sexp          =  c-sexp / b-base-64

b-base-64       =  "{" *whitespace *base-64-chars base-64-end "}"
                     ; encodes a c-sexp, which has a minimum
                     ; length of 2
```

8. Restricted S-Expressions

This document has described S-expressions in general form. Applications may wish to restrict their use of S-expressions in various ways as well as to specify a different default display-hint. Here are some possible restrictions that might be considered:

- no advanced representations (only canonical and basic)
- no display-hints
- no lengths on hexadecimal, quoted-strings, or base-64 encodings
- no empty lists
- no empty octet-strings
- no lists having another list as its first element
- no base-64 or hexadecimal encodings
- fixed limits on the size of octet-strings

As provided in Section 6, conformant implementations will support canonical and basic representation, but support for advanced representation is not generally required. Thus, advanced representation can only be used in applications that mandate its support or where a capability discovery mechanism indicates support.

9. In-Memory Representations

For processing, the S-expression would typically be parsed and represented in memory in a way that is more amenable to efficient processing. This document suggests two alternatives:

- "list-structure"
- "array-layout"

These are only sketched here, as they are only suggestive. The code in [SexpCode] illustrates these styles in more detail.

9.1. List-Structure Memory Representation

Here there are separate records for simple-strings, strings, and lists or list nodes. An S-expression of the form ("abc" "de") could be encoded as two records for the simple-strings, two for the strings, and two for the list elements where a record is a relatively small block of memory and, except for simple-string, might have pointers in it to other records. This is a fairly conventional representation as discussed in Section 4 of [LISP2].

9.2. Array-Layout Memory Representation

Here each S-expression is represented as a contiguous array of octets. The first octet codes the "type" of the S-expression:

```
01   octet-string
```

```
02   octet-string with display-hint
```

```
03   beginning of list (and 00 is used for "end of list")
```

Each of the three types is immediately followed by a k-octet integer indicating the size (in octets) of the following representation. Here, k is an integer that depends on the implementation. It might be anywhere from 2 to 8, but it would be fixed for a given implementation; it determines the size of the objects that can be handled. The transport and canonical representations are independent of the choice of k made by the implementation.

Although the lengths of lists are not given in the usual S-expression notations, it is easy to fill them in when parsing; when you reach a right parenthesis, you know how long the list representation was and where to go back to fill in the missing length.

9.2.1. Octet-String

This is represented as follows:

```
    01 <length> <octet-string>
```

For example (here, k = 2):

```
    01 0003 a b c
```

9.2.2. Octet-String with Display-Hint

This is represented as follows:

```
02 <length>
  01 <length> <octet-string>    /* for display-type */
  01 <length> <octet-string>    /* for octet-string */
```

For example, the S-expression:

```
[gif] #61626364#
```

would be represented as (with k = 2):

```
02 000d
  01 0003  g  i  f
  01 0004 61 62 63 64
```

9.2.3. List

This is represented as:

```
03 <length> <item1> <item2> <item3> ... <item> 00
```

For example, the list (abc [d]ef (g)) is represented in memory as (with k = 2):

```
03 001b
  01 0003 a b c
  02 0009
    01 0001 d
    01 0002 e f
  03 0005
    01 0001 g
  00
00
```

10. Security Considerations

As a pure data representation format, there are few security considerations to S-expressions. A canonical form is required for the consistent creation and verification of digital signatures. This is provided in Section 6.2.

The default display-hint (see Section 4.6) can be specified for an application. Note that if S-expressions containing untyped octet-strings represented for that application are processed by a different application, those untyped octet-string may be treated as if they had a different display-hint.

11. IANA Considerations

This document has no IANA actions.

12. References

12.1. Normative References

[C88] Kernighan, B. and D. Ritchie, "The C Programming Language", ISBN 0-13-110370-9, 1988.

[RFC0020] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <https://www.rfc-editor.org/info/rfc20>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

12.2. Informative References

[BERN] Bernstein, D. J., "Netstrings", Work in Progress, Internet-Draft, draft-bernstein-netstrings-02, 1 January 1997, <https://datatracker.ietf.org/doc/html/draft-bernstein-netstrings-02>.

[CANON1] Wikipedia, "Canonical S-expressions", <https://en.wikipedia.org/wiki/Canonical_S-expressions>.

[CANON2] Grinberg, R., "Csexp - Canonical S-expressions", 24 March 2023, <https://github.com/ocaml-dune/csexp>.

[CDDL-freezer] Bormann, C., "A feature freezer for the Concise Data Definition Language (CDDL)", Work in Progress, Internet-Draft, draft-bormann-cbor-cddl-freezer-15, 28 February 2025, <https://datatracker.ietf.org/doc/html/draft-bormann-cbor-cddl-freezer-15>.

[Formal] Petit-Huguenin, M., "A Formalization of Symbolic Expressions", Work in Progress, Internet-Draft, draft-petithuguenin-ufmrg-formal-sexpr-06, 4 May 2025, <https://datatracker.ietf.org/doc/html/draft-petithuguenin-ufmrg-formal-sexpr-06>.

[GnuPG] GnuPG, "The GNU Privacy Guard", <https://www.gnupg.org/>.

[Inferno] "Inferno S-expressions", Inferno Manual Page, <https://man.cat-v.org/inferno/6/sexprs>.

[Libgcrypt] GnuPG, "The Libgcrypt Library", Libgcrypt version 1.10.2, 6 April 2023, <https://www.gnupg.org/documentation/manuals/gcrypt/>.

[LISP] McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., and M. Levin, "LISP 1.5 Programmer's Manual", ISBN-13 978-0-262-12011-0, ISBN-10 0262130114, 15 August 1962, <https://www.softwarepreservation.org/projects/LISP/book/LISP%201.5%20Programmers%20Manual.pdf>.

[LISP2] McCarthy, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I", April 1960, <https://people.cs.umass.edu/~emery/classes/cmpsci691st/readings/PL/LISP.pdf>.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <https://www.rfc-editor.org/info/rfc2046>.

[RFC2130] Weider, C., Preston, C., Simonsen, K., Alvestrand, H., Atkinson, R., Crispin, M., and P. Svanberg, "The Report of the IAB Character Set Workshop held 29 February - 1 March, 1996", RFC 2130, DOI 10.17487/RFC2130, April 1997, <https://www.rfc-editor.org/info/rfc2130>.

[RFC2692] Ellison, C., "SPKI Requirements", RFC 2692, DOI 10.17487/RFC2692, September 1999, <https://www.rfc-editor.org/info/rfc2692>.

[RFC2693] Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., and T. Ylonen, "SPKI Certificate Theory", RFC 2693, DOI 10.17487/RFC2693, September 1999, <https://www.rfc-editor.org/info/rfc2693>.

[RFC3259] Ott, J., Perkins, C., and D. Kutscher, "A Message Bus for Local Coordination", RFC 3259, DOI 10.17487/RFC3259, April 2002, <https://www.rfc-editor.org/info/rfc3259>.

[RFC3275] Eastlake 3rd, D., Reagle, J., and D. Solo, "(Extensible Markup Language) XML-Signature Syntax and Processing", RFC 3275, DOI 10.17487/RFC3275, March 2002, <https://www.rfc-editor.org/info/rfc3275>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

[Ribose] Ribose Group Inc., "Open-source projects for developers and designers", <https://open.ribose.com/>.

[RNPGP_SEXPP] "S-Expressions parser and generator library in C++ (SEXP in C++)", Version 0.9.2, commit 249c6e3, 22 March 2025, <https://github.com/rnpgp/sexpp>.

[SDSI] Rivest, R. and B. Lampson, "A Simple Distributed Security Architecture", Working document for SDSI version 1.1, 2 October 1996, <https://people.csail.mit.edu/rivest/pubs/RL96.ver-1.1.html>.

[SexpCode] "SEXP---(S-expressions)", commit 4aa7c36, 10 June 2015, <https://github.com/jpmalkiewicz/rivest-sexp>.

[SEXPP] "SexpProcessor", commit a90f90f, 11 April 2025, <https://github.com/seattlerb/sexp_processor>.

[SFEXP] "Small Fast X-Expression Library", commit b7d3bea, 24 March 2023, <https://github.com/mjsottile/sfsexp>.

[SPKI] Rivest, R., "SPKI/SDSI 2.0 A Simple Distributed Security Infrastructure", <https://people.csail.mit.edu/rivest/pubs/RL96.slides-maryland.pdf>.

[Unicode] The Unicode Consortium, "The Unicode Standard", <https://www.unicode.org/versions/latest/>.

[XML] Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., and F. Yergeau, "Extensible Markup Language (XML) 1.0", W3C Recommendation, 26 November 2008, <https://www.w3.org/TR/2008/REC-xml-20081126/>. Latest version available at <https://www.w3.org/TR/REC-xml/>.

Appendix A. Implementations

At this time there are multiple implementations, many open source, available that are intended to read and parse some or all of the various S-expression formats specified here. In particular, see the following -- likely incomplete -- list:

- Project GNU's [Libgcrypt]
- Ribose's RNP [RNPGP_SEXPP] in C++
- Github project of J. P. Malkiewicz [SexpCode] in C
- The Inferno implementation [Inferno]
- Small Fast X-Expression Library [SFEXP]

- S-expression Processor [SEXPP] in Ruby
- Canonical S-expressions [CANON2] (OCAML)

Acknowledgements

Special thanks to Daniel K. Gillmor for his extensive comments.

The comments and suggestions of the following are gratefully acknowledged: John Klensin and Caleb Malchik.

Contributors

Special thanks to Marc Petit-Huguenin, particularly for his extensive work and advice on the ABNF and on locating and fixing unclear parts of earlier draft versions of this document:

Marc Petit-Huguenin
Impedance Mismatch LLC
Email: marc@petit-huguenin.org

Authors' Addresses

Ronald L. Rivest
MIT CSAIL
32 Vassar Street, Room 32-G692
Cambridge, Massachusetts 02139
United States of America
Email: rivest@mit.edu
URI: https://www.csail.mit.edu/person/ronald-l-rivest

Donald E. Eastlake 3rd
Independent
2386 Panoramic Circle
Apopka, Florida 32703
United States of America
Phone: +1-508-333-2270
Email: d3e3e3@gmail.com