Network Working Group
Internet-Draft
Intended status: Informational
Expires: 5 November 2025

M. Petit-Huguenin
Impedance Mismatch LLC

4 May 2025

A Formalization of Symbolic Expressions
draft-petithuguenin-ufmrg-formal-sexpr-06

Abstract

The goal of this document is to show and explain the formal model developed to guarantee that the examples and ABNF in the "SPKI Symbolic Expressions" Internet-Draft are correct.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 5 November 2025.

Table of Contents

1. Introduction

> Mathematics is nature's way of letting you know how sloppy your writing is.
>
> -- Leslie Lamport

A Computerate Specification [ComputerateSpecification] is a mix of a formal and an informal specification, where parts of the informal specification are generated from the formal part. The formal specification is then erased when generating an Internet-Draft for the IETF or a Confluence page for enterprises.

SPKI Symbolic Expressions [SPKI-SExpr] is a specification for symbolic expressions ("s-expr") that is the result of editing a specification originally written back in 1996 by Ronald Rivest. This is done for the purpose of publishing it as an RFC and thus getting a stable reference.

This document shows and explains the formal specification as if that editing was done as a computerate specification. It is not an analysis and formalization of [SPKI-SExpr], but rather the justification for some of its modifications.

This document uses the programming language [Idris2] as a formal method to build a formal model for s-expr that is sound and complete, and to build and verify proofs of that model. As a result the whole text of this document is interspersed with Idris2 code (something called literate programming), which can be extracted and verified as explained in Appendix A.

Because the original document is no longer available on-line, the relevant parts are quoted instead of being referenced, using the recommendations in [RFC8792] to wrap long lines.

2. Terminology

The following terminology defines some mathematical terms that are not used often at the IETF. These terms may have different definitions outside of this document, but only the definitions listed here are relevant in the context of this document:

Completeness: describes a formal system that accepts all valid strings

Curry-Howard Isomorphism: a relation that expresses the fact that computer programs and mathematical proofs are the same thing

Formal Language: a language that have explicit syntax and semantics

Formal Method: the combination of a formal language and a verification system

Formal Model: a representation of a system using a formal language

Formal Specification: the specification of a system using formal methods

Isomorphism: the property that two or more structures are carrying the exact same information

Normalization: the simplification of a proof which, in a program, corresponds to code reduction (also known as code execution)

Proof: the concrete evidence for a proposition which, in a program, corresponds to code that type-checks for the type that corresponds to that proposition

Proposition: a mathematical statement in constructive logic which, in a program, corresponds to a type

Soundness: describes a formal system that rejects all invalid strings

Totality: the property of a function that always returns a value in finite time for any possible input

3. Analysis and Formalization of draft-rivest-sexp-00.txt

This whole section is about the original document made available in 1997.

The first subsection of each of the following sections is an analysis of the original document for ambiguities and contradictions, together with the resolution of these ambiguities and contradictions.

The second subsection shows and explains the formal model of what is discussed in the first subsection.

Formalization only guarantees that an instance of the model has exactly the same bugs that its model, so there is a need for a validation of that model. The third subsection shows proofs of correctness for each example in the original document.

First we need to import a module from the Idris2 standard library:

```
module FormalSexpr

import Data.Bits
```

We want Idris2 to fail to type-check the code if totality cannot be verified for any of the functions, which is done which the following pragma:

```
%default total
```

Note that this pragma actually makes Idris2 non-Turing complete for the code in that document.

Our type is indexed over a list of octets, which makes it a dependent type. Per the Curry-Howard isomorphism, this type acts as a proposition that can be read in plain English as "there exists a list of octets that is a valid s-expr".

Idris’s Bits8 is what the IETF calls an octet, so a List Bits8 is a list of octets.

Note that we do not use characters at all in this formalization, as s-expr are not defined for characters but for octets, but is it possible to convert a list of octets into an equivalent characters string in the Idris REPL:

```
Main> pack $ map (chr . cast) [51, 58, 97, 98, 99]
"3:abc"
```

Alternatively the show function can be used directly on a list of octets after loading the code in the REPL:

```
Show (List Bits8) where
  show = pack . map (chr . cast)
```

Our indexed types are actually families of types, one for each possible value of the index of type List Bits8. Because there is an infinite number of values possible for the index, that actually defines an infinite number of types, one for each possible list of octets, each either a valid s-expr or not.

Only the types in that family that are indexed over a list of octets that is a valid s-expr can have an instance that type-checks. Per the Curry-Howard isomorphim, that instance is considered a proof of the corresponding proposition. Conversely the impossibility of finding an instance of a specific type is a proof that the index is not a valid s-expr.

We need a way to extract the underlying octet-string for each representation that we are going to define. This is done by declaring an ad-hoc polymorphic function in an interface:

```
interface OctetString ty where
  octetString : ty -> List Bits8
```

3.1. Verbatim Representation

3.1.1. Analysis

Section 4.1 of the original s-expr document states:

> NOTE: '\\' line wrapping per RFC 8792
>
> A verbatim encoding of an octet string consists of four 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.

There is a slight confusion here between an octet-string and its representation as verbatim, which is understandable in this context because they look exactly the same.

There is no possible BNF that is sound for the verbatim representation.

3.1.2. Formalization

We need first to reimplement the Idris2 function that is used to convert a number into an equivalent list of octets using the ASCII encoding, as we generally cannot use functions that uses primitives in types because they do not reduce:

```
base10 : Nat -> List Bits8
base10 0 = [48]
base10 x = base' [] x where
  base' : List Bits8 -> Nat -> List Bits8
  base' xs 0 = xs
  base' xs n =
    let (d, m) = divmodNatNZ n 10 ItIsSucc
        m' = cast (m + 0x30)
    in assert_total $ base' (m' :: xs) d
```

Then the Verbatim type is defined for the verbatim representation of an octet-string:

```
data Verbatim : List Bits8 -> Type where
  MkVerbatim : (xs : List Bits8) ->
    Verbatim (base10 (length xs) ++ [58] ++ xs)
```

Then we define the octetString function for the Verbatim type:

```
OctetString (Verbatim _) where
  octetString (MkVerbatim xs) = xs
```

3.1.3. Validation

Idris2 is expressive enough to allow embedding unit tests in the same source and run them as part of the type-checking.

Here we prove that all the examples in section 4.1 of the original document are valid instances of the Verbatim type:

Here are some sample verbatim encodings:

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

* 3:abc

```
testVerbatim1 : Verbatim [51, 58, 97, 98, 99]
testVerbatim1 = MkVerbatim [97, 98, 99]
```

* 7:subject

```
testVerbatim2 : Verbatim [55, 58, 115, 117, 98, 106, 101, 99,
  116]
testVerbatim2 = MkVerbatim [115, 117, 98, 106, 101, 99, 116]
```

* 4:::::

```
testVerbatim3 : Verbatim [52, 58, 58, 58, 58, 58]
testVerbatim3 = MkVerbatim [58, 58, 58, 58]
```

* 12:hello world!

```
testVerbatim4 : Verbatim [49, 50, 58, 104, 101, 108, 108, 111,
  32, 119, 111, 114, 108, 100, 33]
testVerbatim4 = MkVerbatim [104, 101, 108, 108, 111, 32, 119,
  111, 114, 108, 100, 33]
```

* 10:abcdefghij

```
testVerbatim5 : Verbatim [49, 48, 58, 97, 98, 99, 100, 101,
  102, 103, 104, 105, 106]
testVerbatim5 = MkVerbatim [97, 98, 99, 100, 101, 102, 103,
  104, 105, 106]
```

* 0:

```
testVerbatim6 : Verbatim [48, 58]
testVerbatim6 = MkVerbatim []
```

3.2. Quoted-String Representation

3.2.1. Analysis

Section 4.2 of the original s-expr document states:

NOTE: '\\' line wrapping per RFC 8792

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

-- an optional decimal length field

-- an initial double-quote (")

-- the octet string with "C" escape conventions (\n,etc)

-- a final double-quote (")

The specified length is the length of the resulting string after \
\any
escape sequences have been handled. The string does not have any
"terminating NULL" that C includes, and the length does not \
\count such
a character.

The length is optional.

There is no possible BNF that is sound for the quoted-string representation when preceded with the length.

Section 4.2 continues with:

```
      NOTE: '\\' line wrapping per RFC 8792

      The escape conventions within the quoted string are as follows \
      \(these follow
      the "C" programming language conventions, with an extension for
      ignoring line terminators of just LF or CRLF):
              \b              -- backspace
              \t              -- horizontal tab
              \v              -- vertical tab
              \n              -- new-line
              \f              -- form-feed
              \r              -- carriage-return
              \"              -- double-quote
              \'              -- single-quote
              \\              -- 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 linefeed to be ignored
              \<carriage-return><line-feed> -- causes CRLF to be \
      \ignored.
              \<line-feed><carriage-return> -- causes LFCR to be \
      \ignored.
```

Here the first sentence does not match the list of line terminators below it. We assume that there are four line terminators, not two.

In C the escape sequence '\0' is defined for character, but not for strings as a 0 value can never appear in a C string. But that is not true of a quoted-string and it would even be useful to have a shorter encoding than "\x00". We assume that is was not an oversight, and do not add "\0" as escape sequence.

3.2.2. Formalization

There is between four and seven different ways to represent an octet in a quoted-string: ASCII, escaped, octal, and hexadecimal, with that last one taking up to 4 different different representations depending on the combination of uppercase and lowercase symbols.

We first define a function for each of the different type of encodings that returns either a non-empty list of octets if the octet is representable in that encoding, or an empty list if it is not:

* The octets of value 32, 33, 35-91, and 93-126 can be represented as the equivalent ASCII character:

```
ascii : Bits8 -> List Bits8
ascii m =
  if m < 32 then empty
  else if m == 34 then empty
  else if m == 92 then empty
  else if m > 126 then empty
  else [m]
```

* The octets of value 7, 8, 9, 10, 11, 12, 13, 34, 39, and 92 can be represented respectively as the ASCII sequences "\a", "\b", "\t", "\n", "\v", "\f", "\r", "\"", "\'", and "\\":

```
escaped : Bits8 -> List Bits8
escaped 7 = [92, 97]
escaped 8 = [92, 98]
escaped 9 = [92, 116]
escaped 10 = [92, 110]
escaped 11 = [92, 118]
escaped 12 = [92, 102]
escaped 13 = [92, 114]
escaped 34 = [92, 34]
escaped 39 = [92, 39]
escaped 92 = [92, 92]
escaped _ = empty
```

* All octets can be represented as the "\" ASCII character followed by the octal encoding of that octet in ASCII:

```
octal : Bits8 -> List Bits8
octal x =
  let m = x `shiftR` 6
      n = (x `shiftR` 3) .&. 7
      o = x .&. 7
  in [92, m + 48, n + 48, o + 48]
```

* All octets can be represented as the "\x" ASCII sequence followed by the hexadecimal encoding of that octet. Because alphabetic hexadecimal symbols can be encoded as lowercase or uppercase symbols, we get two different encodings for each half of an octet:

```
halfl : Bits8 -> Bits8
halfl x = if x < 10 then x + 48 else x + 87

halfu : Bits8 -> Bits8
halfu x = if x < 10 then x + 48 else x + 55
```

Which then gives us four different hexadecimal encodings for an octet:

```
hexll : Bits8 -> List Bits8
hexll x = [92, 120, halfl (x `shiftR` 4), halfl (x .&. 15)]

hexlu : Bits8 -> List Bits8
hexlu x = [92, 120, halfl (x `shiftR` 4), halfu (x .&. 15)]

hexul : Bits8 -> List Bits8
hexul x = [92, 120, halfu (x `shiftR` 4), halfl (x .&. 15)]

hexuu : Bits8 -> List Bits8
hexuu x = [92, 120, halfu (x `shiftR` 4), halfu (x .&. 15)]
```

We then define a Quoted type indexed over the quoted-string representation of a single octet, using one constructor for each possible type of representation for an octet.

A boolean expression is used to restrict the possible values of the octet when encoded as an ASCII or escaped value, preventing the corresponding constructors to be instantiated.

We also have four additional constructors for the four types of line breaks. These are purely cosmetic and do not encode an octet.

```
data Quoted : List Bits8 -> Type where
  Ascii :
    (x : Bits8) -> (prf : (x >= 32 && x <= 127 && x /= 34 &&
    x /= 92) === True) ->
    Quoted (ascii x)
  Escaped : (x : Bits8) ->
    (prf : (x >= 7 && x <= 13
     || x == 34 || x == 39 || x == 92) === True) ->
    Quoted (escaped x)
  HexLL : (x : Bits8) -> Quoted (hexll x)
  HexUL : (x : Bits8) -> Quoted (hexul x)
  HexLU : (x : Bits8) -> Quoted (hexlu x)
  HexUU : (x : Bits8) -> Quoted (hexuu x)
  Octal : (x : Bits8) -> Quoted (octal x)
  Cr : Quoted [92, 13]
  Lf : Quoted [92, 10]
  CrLf : Quoted [92, 13, 10]
  LfCr : Quoted [92, 10, 13]
```

We can then use that type to build a type indexed over a complete quoted-string. Here we use an Idris2 namespace so we can use the syntactic sugar for a list multiple times in the same source:

```
namespace QuotedString
  public export
  data QuotedList : List Bits8 -> Type where
    Nil : QuotedList []
    (::) : Quoted xs -> QuotedList ys ->
      QuotedList (xs ++ ys)
```

We can then define the octetString function for the QuotedList type:

```
OctetString (QuotedList _) where
  octetString [] = []
  octetString (Ascii x _ :: y) = x :: octetString y
  octetString (Escaped x _ :: y) = x :: octetString y
  octetString (HexLL x :: y) = x :: octetString y
  octetString (HexUL x :: y) = x :: octetString y
  octetString (HexLU x :: y) = x :: octetString y
  octetString (HexUU x :: y) = x :: octetString y
  octetString (Octal x :: y) = x :: octetString y
  octetString (Cr :: y) = octetString y
  octetString (Lf :: y) = octetString y
  octetString (CrLf :: y) = octetString y
  octetString (LfCr :: y) = octetString y
```

The type for a quoted-string:

```
data QuotedString : List Bits8 -> Type where
  MkQuotedString : QuotedList xs ->
    QuotedString (34 :: xs ++ [34])
```

And the function to retrieve its octet-string:

```
OctetString (QuotedString _) where
  octetString (MkQuotedString q) = octetString q
```

We then define an alternative type for the quoted-string representation that is preceded by the length of its octet-string:

```
data QuotedStringLength : List Bits8 -> Type where
  MkQuotedStringLength : (q : QuotedList xs) ->
    QuotedStringLength (base10 (length (octetString q)) ++
      [34] ++ xs ++ [34])
```

And the function to retrieve its octet-string:

```
OctetString (QuotedStringLength _) where
  octetString (MkQuotedStringLength q) = octetString q
```

3.2.3. Validation

Here we prove that all the examples in section 4.2 of the original document are valid instances of the QuotedString or QuotedStringLength types:

Here are some examples of quoted-string encodings:

```
"subject"
"hi there"
7"subject"
3"\n\n\n"
"This has\n two lines."
"This has\
one."
""
```

* "subject"

```
testQuotedString1 : QuotedString [34, 115, 117, 98, 106, 101,
  99, 116, 34]
testQuotedString1 = MkQuotedString [Ascii 115 Refl,
  Ascii 117 Refl, Ascii 98 Refl, Ascii 106 Refl,
  Ascii 101 Refl, Ascii 99 Refl, Ascii 116 Refl]
```

* "hi there"

```
testQuotedString2 : QuotedString [34, 104, 105, 32, 116, 104,
  101, 114, 101, 34]
testQuotedString2 = MkQuotedString [Ascii 104 Refl,
  Ascii 105 Refl, Ascii 32 Refl, Ascii 116 Refl,
  Ascii 104 Refl, Ascii 101 Refl, Ascii 114 Refl,
  Ascii 101 Refl]
```

* 7"subject"

```
testQuotedString3 : QuotedStringLength [55, 34, 115, 117, 98,
  106, 101, 99, 116, 34]
testQuotedString3 = MkQuotedStringLength [Ascii 115 Refl,
  Ascii 117 Refl, Ascii 98 Refl, Ascii 106 Refl,
  Ascii 101 Refl, Ascii 99 Refl, Ascii 116 Refl]
```

* 3"\n\n\n"

```
testQuotedString4 : QuotedStringLength [51, 34, 92, 110, 92,
  110, 92, 110, 34]
testQuotedString4 = MkQuotedStringLength [Escaped 10 Refl,
  Escaped 10 Refl, Escaped 10 Refl]
```

* "This has\n two lines."

```
testQuotedString5 : QuotedString [34, 84, 104, 105, 115, 32,
  104, 97, 115, 92, 110, 32, 116, 119, 111, 32, 108, 105,
  110, 101, 115, 46, 34]
testQuotedString5 = MkQuotedString [Ascii 84 Refl,
  Ascii 104 Refl, Ascii 105 Refl, Ascii 115 Refl,
  Ascii 32 Refl, Ascii 104 Refl, Ascii 97 Refl, Ascii 115 Refl,
  Escaped 10 Refl, Ascii 32 Refl, Ascii 116 Refl,
  Ascii 119 Refl, Ascii 111 Refl, Ascii 32 Refl,
  Ascii 108 Refl, Ascii 105 Refl, Ascii 110 Refl,
  Ascii 101 Refl, Ascii 115 Refl, Ascii 46 Refl]
```

* "This has\ one." (actually on two lines)

```
testQuotedString6 : QuotedString [34, 84, 104, 105, 115, 32,
  104, 97, 115, 92, 10, 111, 110, 101, 46, 34]
testQuotedString6 = MkQuotedString [Ascii 84 Refl,
  Ascii 104 Refl, Ascii 105 Refl, Ascii 115 Refl,
  Ascii 32 Refl, Ascii 104 Refl, Ascii 97 Refl,
  Ascii 115 Refl, Lf, Ascii 111 Refl, Ascii 110 Refl,
  Ascii 101 Refl, Ascii 46 Refl]
```

* ""

```
testQuotedString7 : QuotedString [34, 34]
testQuotedString7 = MkQuotedString []
```

3.3. Token Representation

3.3.1. Analysis

Section 4.3 of the original s-expr document states:

```
      NOTE: '\\' line wrapping per RFC 8792

      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 eight "pseudo-alphabetic" \
      \punctuation marks:
                              -   .   /   _   :  *  +  =
              (Note: upper and lower case are not equivalent.)
              (Note: A token may begin with punctuation, including ":").
```

3.3.2. Formalization

At the difference of all the other encodings, a token element can represent only a subset of all possible octets so we first define a type that constrains any octets but the first in a token:

```
data TokenChar : Bits8 -> Type where
  MkTokenChar : (x : Bits8) ->
    (prf : (x >= 65 && x <= 90 || x >= 97 && x <= 122 ||
      x >= 48 && x <= 57 || x == 45 || x == 46 || x == 47 ||
      x == 95 || x == 58 || x == 42 || x == 43 || x == 61)
      === True) ->
     TokenChar x
```

Then we define a type for a list of these:

```
namespace Token
  public export
  data TokenCharList : List Bits8 -> Type where
    Nil : TokenCharList []
    (::) : TokenChar x -> TokenCharList xs ->
    TokenCharList (x :: xs)
```

Then a type that represents a complete token as a constrained first octet followed by a list of constrained octets:

```
data Token : List Bits8 -> Type where
  MkToken : (x : Bits8) ->
    (prf : (x >= 65 && x <= 90 || x >= 97 && x <= 122 ||
      x == 45 || x == 46 || x == 95 || x == 58 || x == 42 ||
      x == 47 || x == 43 || x == 61) === True) ->
    TokenCharList xs -> Token (x :: xs)
```

We can then define the octetString function for the Token type:

```
OctetString (Token _) where
  octetString (MkToken x _ xs) = x :: octetString' xs where
    octetString' : TokenCharList _ -> List Bits8
    octetString' [] = []
    octetString' (MkTokenChar x _ :: xs) = x :: octetString' xs
```

3.3.3. Validation

Here we prove that all the examples in section 4.3 of the original document are valid instances of the Token type:

Here are some examples of token representations:

```
        subject
        not-before
        class-of-1997
        //microsoft.com/names/smith
        *
```

* subject

```
testToken1 : Token [115, 117, 98, 106, 101, 99, 116]
testToken1 = MkToken 115 Refl [MkTokenChar 117 Refl,
  MkTokenChar 98 Refl, MkTokenChar 106 Refl,
  MkTokenChar 101 Refl, MkTokenChar 99 Refl,
  MkTokenChar 116 Refl]
```

* not-before

```
testToken2 : Token [110, 111, 116, 45, 98, 101, 102, 111, 114,
  101]
testToken2 = MkToken 110 Refl [MkTokenChar 111 Refl,
  MkTokenChar 116 Refl, MkTokenChar 45 Refl,
  MkTokenChar 98 Refl, MkTokenChar 101 Refl,
  MkTokenChar 102 Refl, MkTokenChar 111 Refl,
  MkTokenChar 114 Refl, MkTokenChar 101 Refl]
```

* class-of-1997

```
testToken3 : Token [99, 108, 97, 115, 115, 45, 111, 102, 45,
  49, 57, 57, 55]
testToken3 = MkToken 99 Refl [MkTokenChar 108 Refl,
  MkTokenChar 97 Refl, MkTokenChar 115 Refl,
  MkTokenChar 115 Refl, MkTokenChar 45 Refl,
  MkTokenChar 111 Refl, MkTokenChar 102 Refl,
  MkTokenChar 45 Refl, MkTokenChar 49 Refl,
  MkTokenChar 57 Refl, MkTokenChar 57 Refl,
  MkTokenChar 55 Refl]
```

* //microsoft.com/names/smith

```
testToken4 : Token [47, 47, 109, 105, 99, 114, 111, 115, 111,
  102, 116, 46, 99, 111, 109, 47, 110, 97, 109, 101, 115, 47,
  115, 109, 105, 116, 104]
testToken4 = MkToken 47 Refl [MkTokenChar 47 Refl,
  MkTokenChar 109 Refl, MkTokenChar 105 Refl,
  MkTokenChar 99 Refl, MkTokenChar 114 Refl,
  MkTokenChar 111 Refl, MkTokenChar 115 Refl,
  MkTokenChar 111 Refl, MkTokenChar 102 Refl,
  MkTokenChar 116 Refl, MkTokenChar 46 Refl,
  MkTokenChar 99 Refl, MkTokenChar 111 Refl,
  MkTokenChar 109 Refl, MkTokenChar 47 Refl,
  MkTokenChar 110 Refl, MkTokenChar 97 Refl,
  MkTokenChar 109 Refl, MkTokenChar 101 Refl,
  MkTokenChar 115 Refl, MkTokenChar 47 Refl,
  MkTokenChar 115 Refl, MkTokenChar 109 Refl,
  MkTokenChar 105 Refl, MkTokenChar 116 Refl,
  MkTokenChar 104 Refl]
```

* *

```
testToken5 : Token [42]
testToken5 = MkToken 42 Refl []
```

3.4. Hexadecimal Representation

3.4.1. Analysis

Section 4.4 of the original s-expr document states:

```
      NOTE: '\\' line wrapping per RFC 8792

      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.

              -- a 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.
```

There is no possible BNF that is sound for the hexadecimal representation when preceded with the length.

"hexadecimal encoding" is understood as allowing either case for each hexadecimal half of the encoding for a single octet.

3.4.2. Formalization

The hexadecimal representation encodes each octet of an octet-string as two octets in ASCII, each followed by zero or more white spaces.

First we built a type for a white space:

```
data Whitespace : Bits8 -> Type where
  MkWhitespace : (x : Bits8) ->
    (prf : (x == 32 || x == 9 || x == 11 ||
      x == 12 || x == 13 || x == 10) === True) ->
    Whitespace x
```

And then a type for a list of white spaces:

```
namespace Whitespace
  public export
  data WhitespaceList : List Bits8 -> Type where
    Nil : WhitespaceList []
    (::) : Whitespace x -> WhitespaceList xs ->
      WhitespaceList (x :: xs)
```

Note that white spaces are purely cosmetic, so they do not encode octets in an octet-string. That means that there no octetString function for these.

With that we can build the hexadecimal representation of an octet. We have four constructors, each corresponding to one of the four possible variants for an hexadecimal encoding:

```
data Hex : List Bits8 -> Type where
  HexLL' : (x : Bits8) -> WhitespaceList xs ->
    WhitespaceList ys ->
    Hex ([halfl (x `shiftR` 4)] ++ xs ++ [halfl (x .&. 15)] ++
    ys)
  HexLU' : (x : Bits8) -> WhitespaceList xs ->
    WhitespaceList ys ->
    Hex ([halfl (x `shiftR` 4)] ++ xs ++ [halfu (x .&. 15)] ++
     ys)
  HexUL' : (x : Bits8) -> WhitespaceList xs ->
    WhitespaceList ys ->
    Hex ([halfu (x `shiftR` 4)] ++ xs ++ [halfl (x .&. 15)] ++
      ys)
  HexUU' : (x : Bits8) -> WhitespaceList xs ->
    WhitespaceList ys ->
    Hex ([halfu (x `shiftR` 4)] ++ xs ++ [halfu (x .&. 15)] ++
      ys)
```

Then we can build a type for the hexadecimal representation of an octet-string:

```
namespace Hexadecimal
  public export
  data HexList : List Bits8 -> Type where
    Nil : HexList []
    (::) : Hex xs -> HexList ys -> HexList (xs ++ ys)
```

And a function octetString for that type:

```
OctetString (HexList _) where
  octetString [] = []
  octetString (HexLL' x _ _ :: xs) = x :: octetString xs
  octetString (HexLU' x _ _ :: xs) = x :: octetString xs
  octetString (HexUL' x _ _ :: xs) = x :: octetString xs
  octetString (HexUU' x _ _ :: xs) = x :: octetString xs
```

With that we can build an Hexadecimal type:

```
data Hexadecimal : List Bits8 -> Type where
  MkHexadecimal : WhitespaceList xs -> HexList ys ->
    Hexadecimal (35 :: xs  ++ ys ++ [35])
```

And its octetString function:

```
OctetString (Hexadecimal _) where
  octetString (MkHexadecimal _ y) = octetString y
```

We then define an alternative type for the hexadecimal representation that is preceded by the length of its octet-string:

```
data HexadecimalLength : List Bits8 -> Type where
  MkHexadecimalLength : WhitespaceList xs ->
    (h : HexList ys) ->
    HexadecimalLength (base10 (length (octetString h)) ++
      [35] ++ xs ++ ys ++ [35])
```

And the function to retrieve its octet-string:

```
OctetString (HexadecimalLength _) where
  octetString (MkHexadecimalLength _ h) = octetString h
```

3.4.3. Validation

Here we prove that all the examples in section 4.4 of the original document are valid instances of the Hexadecimal type:

Here are some examples of hexadecimal encodings:

```
        #616263#               -- represents "abc"
        3#616263#              -- also represents "abc"
        # 616
          263 #                -- also represents "abc"
```

* #616263#

```
testHexadecimal1 : Hexadecimal [35, 54, 49, 54, 50, 54, 51, 35]
testHexadecimal1 = MkHexadecimal [] [HexLL' 97 [] [],
  HexLL' 98 [] [], HexLL' 99 [] []]
```

* 3#616263#

```
testHexadecimal2 : HexadecimalLength [51, 35, 54, 49, 54, 50,
  54, 51, 35]
testHexadecimal2 = MkHexadecimalLength [] [HexLL' 97 [] [],
  HexLL' 98 [] [], HexLL' 99 [] []]
```

* # 616 263 #

```
testHexadecimal3 : Hexadecimal [35, 32, 54, 49, 54, 10, 32, 32,
  50, 54, 51, 32, 35]
testHexadecimal3 = MkHexadecimal [MkWhitespace 32 Refl] [
  HexLL' 97 [] [],
  HexLL' 98 [MkWhitespace 10 Refl, MkWhitespace 32 Refl,
    MkWhitespace 32 Refl] [],
  HexLL' 99 [] [MkWhitespace 32 Refl]]
```

3.5. Base 64 Representation

3.5.1. Analysis

Section 4.5 of the original s-expr document states:

```
NOTE: '\\' line wrapping per RFC 8792

An octet-string may be represented in a base-64 coding \
\consisting of:

        -- an (optional) decimal length of the octet string

        -- a vertical bar "|"

        -- the rfc 1521 base-64 encoding of the octet string.

        -- a final vertical bar "|"

The base-64 encoding uses only the characters
        A-Z  a-z  0-9  +  /  =
It produces four characters of output for each three octets of \
\input.
If the input has one or two left-over octets of input, it \
\produces an
output block of length four ending in two or one equals signs, \
\respectively.
Output routines compliant with this standard MUST output the \
\equals signs
as specified.  Input routines MAY accept inputs where the equals \
\signs are
dropped.

There may be whitespace inserted in the midst of the base-64 \
\encoding
arbitrarily; it is ignored.  It is an error to have characters \
\other
than whitespace and base-64 characters.
```

There is no possible BNF that is sound for the base 64 representation when preceded with the length.

The fragment "...where the equals signs are dropped" is ambiguous as it does not state if it is one or two equals signs that can be dropped, or all equals signs. Here we encode types to support the former interpretation.

3.5.2. Formalization

First we need a function that will return a base 64 octet from the six lower bits of an octet:

```
b64 : Bits8 -> Bits8
b64 x =  if x < 26 then x + 65
  else if x < 52 then x + 71
  else if x < 62 then x - 4
  else if x == 62 then 43
  else if x == 63 then 47
  else 0x3D
```

Next we need four functions that return respectively the first, second, third, and fourth ASCII octet for a group of three octets from the octet-string:

```
b641 : Bits8 -> List Bits8
b641 x1 = [b64 (x1 ‘shiftR‘ 2)]

b642 : Bits8 -> Bits8 -> List Bits8
b642 x1 x2 = [b64 (((x1 .&. 0b11) ‘shiftL‘ 4) .|.
  (x2 ‘shiftR‘ 4))]

b643 : Bits8 -> Bits8 -> List Bits8
b643 x2 x3 = [b64 (((x2 .&. 0b1111) ‘shiftL‘ 2) .|.
  (x3 ‘shiftR‘ 6))]

b644 : Bits8 -> List Bits8
b644 x3 = [b64 (x3 .&. 0b111111)]
```

Our first type for the base 64 representation is for a group of three octets from the octet-string:

```
data Base64Full : List Bits8 -> Type where
  MkBase64Full : (x1 : Bits8) -> (x2 : Bits8) ->
    (x3 : Bits8) ->
    WhitespaceList xs -> WhitespaceList ys ->
    WhitespaceList zs -> WhitespaceList ws ->
    Base64Full (b641 x1 ++ xs ++ b642 x1 x2 ++ ys ++
      b643 x2 x3 ++ zs ++ b644 x3 ++ ws)
```

Then we can build a type for the base 64 representation of an octet-string whose length is a multiple of three:

```
namespace Base64
  public export
  data Base64List : List Bits8 -> Type where
    Nil : Base64List []
    (::) : Base64Full xs -> Base64List ys ->
      Base64List (xs ++ ys)
```

We build another type for the octet-strings that have a length that is not a multiple of three. There is additional constructors to account for the fact that the padding is optional.

```
data Base64End : List Bits8 -> Type where
  EndOnePadPad : (x1 : Bits8) ->
    WhitespaceList xs -> WhitespaceList ys ->
    WhitespaceList zs -> WhitespaceList ws ->
    Base64End (b641 x1 ++ xs ++ b642 x1 0 ++ ys ++ [61] ++ zs
      ++ [61] ++ ws)
  EndOnePad : (x1 : Bits8) ->
    WhitespaceList xs -> WhitespaceList ys ->
    WhitespaceList zs ->
    Base64End (b641 x1 ++ xs ++ b642 x1 0 ++ ys ++ [61] ++ zs)
  EndOne : (x1 : Bits8) ->
    WhitespaceList xs -> WhitespaceList ys ->
    Base64End (b641 x1 ++ xs ++ b642 x1 0 ++ ys)
  EndTwoPad : (x1 : Bits8) -> (x2 : Bits8) ->
    WhitespaceList xs -> WhitespaceList ys ->
    WhitespaceList zs -> WhitespaceList ws ->
    Base64End (b641 x1 ++ xs ++ b642 x1 x2 ++ ys ++ b643 x2 0
      ++ zs ++ [61] ++ ws)
  EndTwo : (x1 : Bits8) -> (x2 : Bits8) ->
    WhitespaceList xs -> WhitespaceList ys ->
    WhitespaceList zs ->
    Base64End (b641 x1 ++ xs ++ b642 x1 x2 ++ ys ++ b643 x2 0
      ++ zs)
```

We then put all these together into a type for base 64 encoding with two constructors, one for octet-strings whose length is a multiple of 3, and one for the others:

```
data Base64' : List Bits8 -> Type where
  Base64Mult3 : Base64List xs -> Base64' xs
  Base64Non : Base64List xs -> Base64End ys ->
    Base64' (xs ++ ys)
```

We can then define the octetString function for the Base64' type:

```
octetString' : Base64List _ -> List Bits8
octetString' [] = []
octetString' (MkBase64Full x1 x2 x3 _ _ _ _ :: xs) =
  x1 :: x2 :: x3 :: octetString' xs

OctetString (Base64' _) where
  octetString (Base64Mult3 xs) = octetString' xs
  octetString (Base64Non xs (EndOnePadPad x1 _ _ _ _)) =
    octetString' xs ++ [x1]
  octetString (Base64Non xs (EndOnePad x1 _ _ _)) =
    octetString' xs ++ [x1]
  octetString (Base64Non xs (EndOne x1 _ _)) =
    octetString' xs ++ [x1]
  octetString (Base64Non xs (EndTwoPad x1 x2 _ _ _ _)) =
    octetString' xs ++ [x1, x2]
  octetString (Base64Non xs (EndTwo x1 x2 _ _ _)) =
    octetString' xs ++ [x1, x2]
```

Finally we can define the Base64 type:

```
data Base64 : List Bits8 -> Type where
  MkBase64 : WhitespaceList xs -> Base64' ys ->
    Base64 (124 :: xs ++ ys ++ [124])
```

And its octetString function:

```
OctetString (Base64 _) where
  octetString (MkBase64 _ y) = octetString y
```

We then reuse the Base64' type to define one more type for the base 64 representation that is preceded by the length of its octet-string:

```
data Base64Length : List Bits8 -> Type where
  MkBase64Length : WhitespaceList xs -> (b : Base64' ys) ->
    Base64Length (base10 (length (octetString b)) ++ [124]
      ++ xs ++ ys ++ [124])
```

And its octetString function:

```
   OctetString (Base64Length _) where
     octetString (MkBase64Length _ b) = octetString b
```

3.5.3. Validation

Here we prove that all the examples in section 4.5 of the original document are valid instances of the Base64 type:

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"
```

* |YWJj|

```
   testBase641 : Base64 [124, 89, 87, 74, 106, 124]
   testBase641 = MkBase64 [] (Base64Mult3
     [MkBase64Full 97 98 99 [] [] [] []])
```

* | Y W J j |

```
   testBase642 : Base64 [124, 32, 89, 32, 87, 32, 74, 32, 106,
     32, 124]
   testBase642 = MkBase64 [MkWhitespace 32 Refl]
     (Base64Mult3 [MkBase64Full 97 98 99 [MkWhitespace 32 Refl]
     [MkWhitespace 32 Refl] [MkWhitespace 32 Refl]
     [MkWhitespace 32 Refl]])
```

* 3|YWJj|

```
   testBase643 : Base64Length [51, 124, 89, 87, 74, 106, 124]
   testBase643 = MkBase64Length [] (Base64Mult3
     [MkBase64Full 97 98 99 [] [] [] []])
```

* |YWJjZA==|

```
   testBase644 : Base64 [124, 89, 87, 74, 106, 90, 65, 61, 61,
     124]
   testBase644 = MkBase64 [] (Base64Non
     [MkBase64Full 97 98 99 [] [] [] []]
     (EndOnePadPad 100 [] [] [] []))
```

* |YWJjZA|

```
testBase645 : Base64 [124, 89, 87, 74, 106, 90, 65, 124]
testBase645 = MkBase64 [] (Base64Non
  [MkBase64Full 97 98 99 [] [] [] []]
  (EndOne 100 [] []))
```

3.6. Octet-String Representation

3.6.1. Analysis

Before going further we have to address the case of the brace notation for base 64.

Section 6.2 of the original s-expr document states:

> NOTE: '\\' line wrapping per RFC 8792
>
> There is a difference between the brace notation for base-64 \
> \used here
> and the || notation for base-64'd octet-strings described above. \
> \ Here
> the base-64 contents are converted to octets, and then \
> \re-scanned as
> if they were given originally as octets. With the || notation, \
> \the
> contents are just turned into an octet-string.

It is not clear from that text if the octets that are to be re-scanned are for the representation of an octet-string, or for a whole s-expr. Additionally this text seems to ignore the fact that examples using that notation were provided in section 2 and section 5 of the original s-expr document.

So the first ambiguity would about about the usage of the brace notation in a display-hint. Obviously it would not make sense to have a s-expr inside a display-hint so at best it encodes an octet-string. But if that's the case, does it encodes any of the other representations (maybe including itself) or just the verbatim representation, as examples in section 2 and 5 show?

The same can be said of the use of the brace notation as simple-string. There again it would not make sense to encode an s-expr with it, because then it would be possible to associate it with a display-hint, which does not make sense. Then if it is only the encoding of the representation of an octet-string then the same ambiguity than above is present about the representations permitted.

To add to the issue, the brace notation for base 64 on an octet-string is largely redundant with the quoted-string, hexadecimal, and base 64 representations, because these already handle the problem of representing any s-expr using ASCII characters. That is only required for the basic transport.

Here we chose to use the brace notation for base 64 exclusively in the basic transport, restricting the octet-string inside as verbatim representations. That makes the examples in section 2 and 5 incorrect unless used as s-expr in the basic transport.

3.6.2. Formalization

With that in mind we can define a type that covers all possible representation for an octet-string, excluding the brace notation for base 64.

```
data Representation : List Bits8 -> Type where
  RepresentationVerbatim : (v : Verbatim xs) ->
    Representation xs
  RepresentationQuoted : QuotedString xs ->
    Representation xs
  RepresentationQuotedLength : QuotedStringLength xs ->
    Representation xs
  RepresentationToken : Token xs -> Representation xs
  RepresentationHexadecimal : Hexadecimal xs ->
    Representation xs
  RepresentationHexadecimalLength : HexadecimalLength xs ->
    Representation xs
  RepresentationBase64 : Base64 xs -> Representation xs
  RepresentationBase64Length : Base64Length xs ->
    Representation xs
```

And its matching octetString function:

```
OctetString (Representation _) where
  octetString (RepresentationVerbatim v) = octetString v
  octetString (RepresentationQuoted x) = octetString x
  octetString (RepresentationQuotedLength x) = octetString x
  octetString (RepresentationToken x) = octetString x
  octetString (RepresentationHexadecimal x) = octetString x
  octetString (RepresentationHexadecimalLength x) =
    octetString x
  octetString (RepresentationBase64 x) = octetString x
  octetString (RepresentationBase64Length x) = octetString x
```

3.7. Display-hint Representation

3.7.1. Analysis

Section 4.6 of the original s-expr document states:

> NOTE: '\\' line wrapping per RFC 8792
>
> Any octet string may be preceded by a single "display hint".
>
> The purposes of the display hint is to provide information on how to display the octet string to a user. It has no other function. Many of the MIME types work here.
>
> A display-hint is an octet string surrounded by square brackets. There may be whitespace separating the octet string from the surrounding brackets. Any of the legal formats may be used for \
> \the
> octet string.

The uses of "octet string" in this fragment are all incorrect. "octet string representation" should be used instead.

The text uses singular "whitespace", not the plural "whitespaces".

The text also does not say if white spaces can separate the display hint from the octet-string it provides information to. We assume that multiple white spaces can be used after the opening bracket, before the closing bracket, and between the closing bracket and the following octet-string.

Following the argument in the argument in the previous section, "legal formats" does not include the brace notation for base 64.

Section 4.6 of the original s-expr document ends with:

> NOTE: '\\' line wrapping per RFC 8792
>
> In applications an octet-string that is untyped may be \
> \considered to have
> a pre-specified "default" mime type. The mime type
> "text/plain; charset=iso-8859-1"
> is the standard default.

3.7.2. Formalization

We first build a type for a display-hint:

```
data DisplayHint : List Bits8 -> Type where
  MkDisplayHint : WhitespaceList xs -> Representation ys ->
    WhitespaceList zs ->
    DisplayHint (91 :: xs ++ ys ++ zs ++ [93])
```

Then define octetString for that type:

```
OctetString (DisplayHint _) where
  octetString (MkDisplayHint _ x _) = octetString x
```

Then a type for the association of a display-hint and the representation of an octet-string:

```
data WithHint : List Bits8 -> Type where
  MkWithHint : DisplayHint xs -> WhitespaceList ys ->
    Representation zs -> WithHint (xs ++ ys ++ zs)
```

We finally define the default display-hint as the token application/octet-stream:

```
defaultHint : Representation [97, 112, 112, 108, 105, 99, 97,
  116, 105, 111, 110, 47, 111, 99, 116, 101, 116, 45, 115,
  116, 114, 101, 97, 109]
defaultHint = RepresentationToken (MkToken 97 Refl
  [MkTokenChar 112 Refl, MkTokenChar 112 Refl,
  MkTokenChar 108 Refl, MkTokenChar 105 Refl,
  MkTokenChar 99 Refl, MkTokenChar 97 Refl,
  MkTokenChar 116 Refl, MkTokenChar 105 Refl,
  MkTokenChar 111 Refl, MkTokenChar 110 Refl,
  MkTokenChar 47 Refl, MkTokenChar 111 Refl,
  MkTokenChar 99 Refl, MkTokenChar 116 Refl,
  MkTokenChar 101 Refl, MkTokenChar 116 Refl,
  MkTokenChar 45 Refl, MkTokenChar 115 Refl,
  MkTokenChar 116 Refl, MkTokenChar 114 Refl,
  MkTokenChar 101 Refl, MkTokenChar 97 Refl,
  MkTokenChar 109 Refl])
```

3.7.3. Validation

Here we prove that all the examples in section 4.6 of the original document are valid instances of the DisplayHint type:

Here are some examples of display-hints:

```
[image/gif]
[URI]
[charset=unicode-1-1]
[text/richtext]
[application/postscript]
[audio/basic]
["http://abc.com/display-types/funky.html"]
```

* [image/gif]

```
testHint1 : DisplayHint [91, 105, 109, 97, 103, 101, 47, 103,
  105, 102, 93]
testHint1 = MkDisplayHint [] (RepresentationToken
  (MkToken 105 Refl [MkTokenChar 109 Refl,
  MkTokenChar 97 Refl, MkTokenChar 103 Refl,
  MkTokenChar 101 Refl, MkTokenChar 47 Refl,
  MkTokenChar 103 Refl, MkTokenChar 105 Refl,
  MkTokenChar 102 Refl])) []
```

* [URI]

```
testHint2 : DisplayHint [91, 85, 82, 73, 93]
testHint2 = MkDisplayHint [] (RepresentationToken
  (MkToken 85 Refl [MkTokenChar 82 Refl,
  MkTokenChar 73 Refl])) []
```

* [charset=unicode-1-1]

```
testHint3 : DisplayHint [91, 99, 104, 97, 114, 115, 101, 116,
  61, 117, 110, 105, 99, 111, 100, 101, 45, 49, 45, 49, 93]
testHint3 = MkDisplayHint [] (RepresentationToken
  (MkToken 99 Refl [MkTokenChar 104 Refl,
  MkTokenChar 97 Refl, MkTokenChar 114 Refl,
  MkTokenChar 115 Refl, MkTokenChar 101 Refl,
  MkTokenChar 116 Refl, MkTokenChar 61 Refl,
  MkTokenChar 117 Refl, MkTokenChar 110 Refl,
  MkTokenChar 105 Refl, MkTokenChar 99 Refl,
  MkTokenChar 111 Refl, MkTokenChar 100 Refl,
  MkTokenChar 101 Refl, MkTokenChar 45 Refl,
  MkTokenChar 49 Refl, MkTokenChar 45 Refl,
  MkTokenChar 49 Refl])) []
```

* [text/richtext]

```
testHint4 : DisplayHint [91, 116, 101, 120, 116, 47, 114, 105,
  99, 104, 116, 101, 120, 116, 93]
testHint4 = MkDisplayHint [] (RepresentationToken
  (MkToken 116 Refl [MkTokenChar 101 Refl,
  MkTokenChar 120 Refl, MkTokenChar 116 Refl,
  MkTokenChar 47 Refl, MkTokenChar 114 Refl,
  MkTokenChar 105 Refl, MkTokenChar 99 Refl,
  MkTokenChar 104 Refl, MkTokenChar 116 Refl,
  MkTokenChar 101 Refl, MkTokenChar 120 Refl,
  MkTokenChar 116 Refl])) []
```

* [application/postscript]

```
testHint5 : DisplayHint [91, 97, 112, 112, 108, 105, 99, 97,
  116, 105, 111, 110, 47, 112, 111, 115, 116, 115, 99, 114,
  105, 112, 116, 93]
testHint5 = MkDisplayHint [] (RepresentationToken
  (MkToken 97 Refl [MkTokenChar 112 Refl,
  MkTokenChar 112 Refl, MkTokenChar 108 Refl,
  MkTokenChar 105 Refl, MkTokenChar 99 Refl,
  MkTokenChar 97 Refl, MkTokenChar 116 Refl,
  MkTokenChar 105 Refl, MkTokenChar 111 Refl,
  MkTokenChar 110 Refl, MkTokenChar 47 Refl,
  MkTokenChar 112 Refl, MkTokenChar 111 Refl,
  MkTokenChar 115 Refl, MkTokenChar 116 Refl,
  MkTokenChar 115 Refl, MkTokenChar 99 Refl,
  MkTokenChar 114 Refl, MkTokenChar 105 Refl,
  MkTokenChar 112 Refl, MkTokenChar 116 Refl])) []
```

* [audio/basic]

```
testHint6 : DisplayHint [91, 97, 117, 100, 105, 111, 47, 98,
  97, 115, 105, 99, 93]
testHint6 = MkDisplayHint [] (RepresentationToken
  (MkToken 97 Refl [MkTokenChar 117 Refl,
  MkTokenChar 100 Refl, MkTokenChar 105 Refl,
  MkTokenChar 111 Refl, MkTokenChar 47 Refl,
  MkTokenChar 98 Refl, MkTokenChar 97 Refl,
  MkTokenChar 115 Refl, MkTokenChar 105 Refl,
  MkTokenChar 99 Refl])) []
```

* ["http://abc.com/display-types/funky.html"]

```
testHint7 : DisplayHint [91, 34, 104, 116, 116, 112, 58, 47,
  47, 97, 98, 99, 46, 99, 111, 109, 47, 100, 105, 115, 112,
  108, 97, 121, 45, 116, 121, 112, 101, 115, 47, 102, 117,
  110, 107, 121, 46, 104, 116, 109, 108, 34, 93]
testHint7 = MkDisplayHint [] (RepresentationQuoted
  (MkQuotedString [Ascii 104 Refl, Ascii 116 Refl,
  Ascii 116 Refl, Ascii 112 Refl, Ascii 58 Refl,
  Ascii 47 Refl, Ascii 47 Refl, Ascii 97 Refl, Ascii 98 Refl,
  Ascii 99 Refl, Ascii 46 Refl, Ascii 99 Refl, Ascii 111 Refl,
  Ascii 109 Refl, Ascii 47 Refl, Ascii 100 Refl,
  Ascii 105 Refl, Ascii 115 Refl, Ascii 112 Refl,
  Ascii 108 Refl, Ascii 97 Refl, Ascii 121 Refl,
  Ascii 45 Refl, Ascii 116 Refl, Ascii 121 Refl,
  Ascii 112 Refl, Ascii 101 Refl, Ascii 115 Refl,
  Ascii 47 Refl, Ascii 102 Refl, Ascii 117 Refl,
  Ascii 110 Refl, Ascii 107 Refl, Ascii 121 Refl,
  Ascii 46 Refl, Ascii 104 Refl, Ascii 116 Refl,
  Ascii 109 Refl, Ascii 108 Refl])) []
```

3.8. Equality of Octet-String

3.8.1. Analysis

Section 4.7 of the original s-expr document states:

> Two octet strings are considered to be "equal" if and only if they have the same display hint and the same data octet strings.
>
> Note that octet-strings are "case-sensitive"; the octet-string \
> \"abc"
> is not equal to the octet-string "ABC".
>
> An untyped octet-string can be compared to another octet-string \
> \(typed
> or not) by considering it as a typed octet-string with the default mime-type.

The term "octet string" here is incorrect as it is described as the combination of a display hint and a "data octet strings", the latter being actually an "octet string representation".

Consequently the terms "equal" or "equality" are incorrect, and the terms "equivalent" or "equivalences" should be used instead. Here the term "equivalent" means "carrying the same information", i.e. the same octet-string. Two octet-string representations can be equivalent, but not equal, e.g, the token abc and the quoted-string "abc" are equivalent but not equal.

The same reasoning is applied when comparing typed octet-string representations, or a typed octet-string representation with an untyped octet-string representation.

3.8.2. Formalization

We first define a type that carries either a typed or an untyped octet-string representation:

```
data Element : Type where
  Untyped : Representation _ -> Element
  Typed : Representation _ -> Representation _ -> Element
```

Then we define the type alias Equivalence as a relation between two elements. Equivalence is already declared in the standard library, so we have to hide that declaration first:

```
%hide Control.Relation.Equivalence

Equivalence : Element -> Element -> Type
Equivalence (Untyped x) (Untyped x') =
  octetString x === octetString x'
Equivalence (Untyped x) (Typed h x') =
  (octetString defaultHint === octetString h,
  octetString x === octetString x')
Equivalence (Typed h x) (Untyped x') =
  (octetString h === octetString defaultHint,
  octetString x === octetString x')
Equivalence (Typed h x) (Typed h' x') =
  (octetString h === octetString h',
  octetString x === octetString x')
```

3.8.3. Validation

Here we prove that a subset of the examples in section 1 of the original document are equivalent. Proving the other equivalences is trivial:

```
      NOTE: '\\' line wrapping per RFC 8792

      An octet-string is a finite sequence of eight-bit octets.  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

              {MzphYmM=}      -- as a base-64 encoding of the verbatim \
      \encoding
                                 (that is, an encoding of "3:abc")

              |YWJj|          -- as a base-64 encoding of the \
      \octet-string "abc"

      These encodings are all equivalent; they all denote the same \
      \octet string.
```

We first proves that the three first representations are correct:

```
   abcToken : Representation [97, 98, 99]
   abcToken = RepresentationToken (MkToken 97 Refl
     [MkTokenChar 98 Refl, MkTokenChar 99 Refl])

   abcQuoted : Representation [34, 97, 98, 99, 34]
   abcQuoted = RepresentationQuoted (MkQuotedString
     [Ascii 97 Refl, Ascii 98 Refl, Ascii 99 Refl])

   abcHex : Representation [35, 54, 49, 54, 50, 54, 51, 35]
   abcHex = RepresentationHexadecimal (MkHexadecimal []
     [HexLL' 97 [] [], HexLL' 98 [] [], HexLL' 99 [] []])
```

We can then prove that abc is equivalent to "abc", and that "abc" is equivalent to #616263#:

```
testEq1 : Equivalence (Untyped FormalSexpr.abcToken)
  (Untyped FormalSexpr.abcQuoted)
testEq1 = Refl

testEq2 : Equivalence (Untyped FormalSexpr.abcQuoted)
  (Untyped FormalSexpr.abcHex)
testEq2 = Refl
```

By transitivity we can then prove that abc is equivalent to #616263#:

```
testEq3 : Equivalence (Untyped FormalSexpr.abcToken)
  (Untyped FormalSexpr.abcHex)
testEq3 = trans testEq1 testEq2
```

We can also use symmetry to prove that if a first octet-string representation is equivalent to a second octet-string representation, then the second is also equivalent to the first one.

```
testEq4 : Equivalence (Untyped FormalSexpr.abcHex)
  (Untyped FormalSexpr.abcToken)
testEq4 = sym testEq3
```

3.9. Lists

3.9.1. Analysis

Section 5 of the original s-expr document states:

> NOTE: '\\' line wrapping per RFC 8792
>
> Just as with octet-strings, there are several ways to represent an S-expression. 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. Also, whitespace may follow the initial left parenthesis, or precede the final right parenthesis.

The first sentence should say that there are different ways to represent a list.

But the issue is really that in some cases the separation between some representations of an octet-string is ambiguous. The actual rules for mandatory separation are:

* a token must be separated from a quoted-string, hexadecimal, or base 64 representation that is prefixed with the length

* a token must be separated from the next token

* a token must be separated from the next verbatim representation

Additionally section 2 states:

> NOTE: '\\' line wrapping per RFC 8792
>
> A list is a finite sequence of zero or more simpler \
> \S-expressions. A list
> may be represented by using parentheses to surround the sequence \
> \of encodings
> of its elements, as in:
>
> (abc (de #6667#) "ghi jkl")

Parentheses are not optional when representing a list, so "may be" should be "are".

3.9.2. Formalization

To represent the various ways to separate representations we need four mutually inductive types, that we first declare as abstract types:

```
data TokenList : List Bits8 -> Type
data SeparateList : List Bits8 -> Type
data OtherList : List Bits8 -> Type
data Lists : List Bits8 -> Type
```

TokenList is the type of a list of octet-string representations that starts with a token:

```
data TokenList : List Bits8 -> Type where
  TokenNil : Token xs -> TokenList xs
  TokenConsToken : Token xs -> Whitespace y ->
    WhitespaceList ys -> TokenList zs ->
    TokenList (xs ++ (y :: ys) ++ zs)
  TokenConsSeparate : Token xs -> Whitespace y ->
    WhitespaceList ys -> SeparateList zs ->
    TokenList (xs ++ (y :: ys) ++ zs)
  TokenConsOther : Token xs -> WhitespaceList ys ->
    OtherList zs -> TokenList (xs ++ ys ++ zs)
```

SeparateList is the type of a list of octet-string representations that starts with an octet-string representation that when inserted after a token will require it to be separated:

```
data SeparateList : List Bits8 -> Type where
  SeparateVerbatim : Verbatim xs -> SeparateList xs
  SeparateVerbatimToken : Verbatim xs -> WhitespaceList ys ->
   TokenList zs -> SeparateList (xs ++ ys ++ zs)
  SeparateVerbatimSeparate : Verbatim xs ->
    WhitespaceList ys -> SeparateList zs ->
    SeparateList (xs ++ ys ++ zs)
  SeparateVerbatimOther : Verbatim xs -> WhitespaceList ys ->
    OtherList zs -> SeparateList (xs ++ ys ++ zs)
  SeparateQuotedStringLength : QuotedStringLength xs ->
    SeparateList xs
  SeparateQuotedStringLengthToken : QuotedStringLength xs ->
    WhitespaceList ys -> TokenList zs ->
    SeparateList (xs ++ ys ++ zs)
  SeparateQuotedStringLengthSeparate : QuotedStringLength xs ->
    WhitespaceList ys -> SeparateList zs ->
    SeparateList (xs ++ ys ++ zs)
  SeparateQuotedStringLengthOther : QuotedStringLength xs ->
    WhitespaceList ys -> OtherList zs ->
    SeparateList (xs ++ ys ++ zs)
  SeparateHexadecimal : HexadecimalLength xs ->
    SeparateList xs
  SeparateHexadecimalLengthToken : HexadecimalLength xs ->
    WhitespaceList ys -> TokenList zs ->
    SeparateList (xs ++ ys ++ zs)
  SeparateHexadecimalLengthSeparate : HexadecimalLength xs ->
    WhitespaceList ys -> SeparateList zs ->
    SeparateList (xs ++ ys ++ zs)
  SeparateHexadecimalLengthOther : HexadecimalLength xs ->
    WhitespaceList ys -> OtherList zs ->
    SeparateList (xs ++ ys ++ zs)
  SeparateBase64 : Base64Length xs ->
    SeparateList xs
  SeparateBase64LengthToken : Base64Length xs ->
    WhitespaceList ys -> TokenList zs ->
    SeparateList (xs ++ ys ++ zs)
  SeparateBase64LengthSeparate : Base64Length xs ->
    WhitespaceList ys -> SeparateList zs ->
    SeparateList (xs ++ ys ++ zs)
  SeparateBase64LengthOther : Base64Length xs ->
    WhitespaceList ys -> OtherList zs ->
    SeparateList (xs ++ ys ++ zs)
```

OtherList is the type of a list of octet-string representations that starts with an octet-string representations that when inserted after a token will not require it to be separated:

```
   data OtherList : List Bits8 -> Type where
     OtherQuotedString : QuotedString xs -> OtherList xs
     OtherQuotedStringToken : QuotedString xs ->
      WhitespaceList ys -> TokenList zs ->
      OtherList (xs ++ ys ++ zs)
     OtherQuotedStringSeparate : QuotedString xs ->
       WhitespaceList ys -> SeparateList zs ->
       OtherList (xs ++ ys ++ zs)
     OtherQuotedStringOther : QuotedString xs ->
       WhitespaceList ys -> OtherList zs ->
       OtherList (xs ++ ys ++ zs)
     OtherHexadecimal : Hexadecimal xs -> OtherList xs
     OtherHexadecimalToken : Hexadecimal xs ->
      WhitespaceList ys -> TokenList zs ->
      OtherList (xs ++ ys ++ zs)
     OtherHexadecimalSeparate : Hexadecimal xs ->
       WhitespaceList ys -> SeparateList zs ->
       OtherList (xs ++ ys ++ zs)
     OtherHexadecimalOther : Hexadecimal xs ->
       WhitespaceList ys -> OtherList zs ->
       OtherList (xs ++ ys ++ zs)
     OtherBase64 : Base64 xs -> OtherList xs
     OtherBase64Token : Base64 xs -> WhitespaceList ys ->
      TokenList zs -> OtherList (xs ++ ys ++ zs)
     OtherBase64Separate : Base64 xs ->
       WhitespaceList ys -> SeparateList zs ->
       OtherList (xs ++ ys ++ zs)
     OtherBase64Other : Base64 xs -> WhitespaceList ys ->
       OtherList zs -> OtherList (xs ++ ys ++ zs)
     OtherHint : WithHint xs -> OtherList xs
     OtherHintToken : WithHint xs ->
      WhitespaceList ys -> TokenList zs ->
      OtherList (xs ++ ys ++ zs)
     OtherHintSeparate : WithHint xs ->
       WhitespaceList ys -> SeparateList zs ->
       OtherList (xs ++ ys ++ zs)
     OtherHintOther : WithHint xs ->
       WhitespaceList ys -> OtherList zs ->
       OtherList (xs ++ ys ++ zs)
     OtherLists : Lists xs -> OtherList xs
     OtherListsToken : Lists xs ->
      WhitespaceList ys -> TokenList zs ->
      OtherList (xs ++ ys ++ zs)
     OtherListsSeparate : Lists xs ->
```

```
          WhitespaceList ys -> SeparateList zs ->
          OtherList (xs ++ ys ++ zs)
        OtherListsOther : Lists xs ->
          WhitespaceList ys -> OtherList zs ->
          OtherList (xs ++ ys ++ zs)
```

And finally the Lists type groups all the possible lists in a s-expr.

```
   data Lists : List Bits8 -> Type where
     ListsTokenList : WhitespaceList xs -> TokenList ys ->
       WhitespaceList zs -> Lists (40 :: xs ++ ys ++ zs ++ [41])
     ListsSeparateList : WhitespaceList xs -> SeparateList ys ->
       WhitespaceList zs -> Lists (40 :: xs ++ ys ++ zs ++ [41])
     ListsOtherList : WhitespaceList xs -> OtherList ys ->
       WhitespaceList zs -> Lists (40 :: xs ++ ys ++ zs ++ [41])
     ListsEmptyList : WhitespaceList xs ->
       Lists (40 :: xs ++ [41])
```

3.9.3. Validation

Here we prove that all the examples in section 5 of the original document except the last one are valid instances of the Lists type:

Here are some examples of encodings of lists:

```
        (a b c)

        ( a ( b c ) ( ( d e ) ( e f ) )  )

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

        ({3Rt=} "1997" murphy 3:{XC++})
```

* (a b c)

```
testLists1 : Lists [40, 97, 32, 98, 32, 99, 41]
testLists1 = ListsTokenList []
  (TokenConsToken (MkToken 97 Refl []) (MkWhitespace 32 Refl)
  [] (TokenConsToken (MkToken 98 Refl [])
  (MkWhitespace 32 Refl) [] (TokenNil (MkToken 99 Refl []))))
  []
```

* (a (b c) ((d e) (e f)))

```
testLists2 : Lists [40, 32, 97, 32, 40, 32, 98, 32, 99, 32, 41,
  32, 40, 32, 40, 32, 100, 32, 101, 32, 41, 32, 40, 32, 101,
  32, 102, 32, 41, 32, 41, 32, 32, 41]
testLists2 = ListsTokenList[MkWhitespace 32 Refl]
  (TokenConsOther (MkToken 97 Refl []) [MkWhitespace 32 Refl]
  (OtherListsOther (ListsTokenList [MkWhitespace 32 Refl]
  (TokenConsToken (MkToken 98 Refl []) (MkWhitespace 32 Refl)
  [] (TokenNil (MkToken 99 Refl [])))) [MkWhitespace 32 Refl])
  [MkWhitespace 32 Refl] (OtherLists (ListsOtherList
  [MkWhitespace 32 Refl] (OtherListsOther (ListsTokenList
  [MkWhitespace 32 Refl] (TokenConsToken (MkToken 100 Refl [])
  (MkWhitespace 32 Refl) [] (TokenNil (MkToken 101 Refl []))))
  [MkWhitespace 32 Refl]) [MkWhitespace 32 Refl] (OtherLists
  (ListsTokenList [MkWhitespace 32 Refl] (TokenConsToken
  (MkToken 101 Refl []) (MkWhitespace 32 Refl) [] (TokenNil
  (MkToken 102 Refl [])))) [MkWhitespace 32 Refl])))
  [MkWhitespace 32 Refl]))))) [MkWhitespace 32 Refl,
  MkWhitespace 32 Refl]
```

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

```
testLists3 : Lists [40, 49, 49, 58, 99, 101, 114, 116, 105,
  102, 105, 99, 97, 116, 101, 40, 54, 58, 105, 115, 115,
  117, 101, 114, 51, 58, 98, 111, 98, 41, 40, 55, 58, 115,
  117, 98, 106, 101, 99, 116, 53, 58, 97, 108, 105, 99,
  101, 41, 41]
testLists3 = ListsSeparateList [] (SeparateVerbatimOther
  (MkVerbatim [99, 101, 114, 116, 105, 102, 105, 99, 97,
  116, 101]) [] (OtherListsOther (ListsSeparateList []
  (SeparateVerbatimSeparate (MkVerbatim [105, 115, 115,
  117, 101, 114]) [] (SeparateVerbatim (MkVerbatim [98,
  111, 98])))) []) [] (OtherLists (ListsSeparateList []
  (SeparateVerbatimSeparate (MkVerbatim [115, 117, 98,
  106, 101, 99, 116]) [] (SeparateVerbatim (MkVerbatim
  [97, 108, 105, 99, 101])))) [])))) []
```

3.10. Advanced S-Expr Transport

3.10.1. Analysis

Section 6.3 of the original s-expr document states:

```
      NOTE: '\\' line wrapping per RFC 8792

      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 \
      \representation
      forms described above, include quoted strings, base-64 and \
      \hexadecimal
      representation of strings, tokens, representations of strings with
      omitted lengths, and so on.
```

Because this transport is aimed at users, we also permit to add white spaces before and after a s-expr.

3.10.2. Formalization

SExpr is the type of advanced transport for valid s-expr:

```
      data SExpr : List Bits8 -> Type where
        SExprRepresentation : WhitespaceList xs ->
          Representation ys -> WhitespaceList zs ->
          SExpr (xs ++ ys ++ zs)
        SExprWithHint : WhitespaceList xs -> WithHint ys ->
          WhitespaceList zs -> SExpr (xs ++ ys ++ zs)
        SExprList : WhitespaceList xs -> Lists ys ->
          WhitespaceList zs -> SExpr (xs ++ ys ++ zs)
```

3.10.3. Validation

Here we prove that the example in section 5 of the original document is a valid instance of the SExpr type:

```
      NOTE: '\\' line wrapping per RFC 8792

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

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

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

```
testSExpr1 : SExpr [40, 97, 98, 99, 32, 40, 100, 101, 32, 35,
  54, 54, 54, 55, 35, 41, 32, 34, 103, 104, 105, 32, 106, 107,
  108, 34, 41]
testSExpr1 = SExprList [] (ListsTokenList [] (TokenConsOther
  (MkToken 97 Refl [MkTokenChar 98 Refl, MkTokenChar 99 Refl])
  [MkWhitespace 32 Refl] (OtherListsOther (ListsTokenList []
  (TokenConsOther (MkToken 100 Refl [MkTokenChar 101 Refl])
  [MkWhitespace 32 Refl] (OtherHexadecimal (MkHexadecimal []
  [HexLL' 102 [] [], HexLL' 103 [][]])))) [])
  [MkWhitespace 32 Refl] (OtherQuotedString (MkQuotedString
  [Ascii 103 Refl, Ascii 104 Refl, Ascii 105 Refl,
  Ascii 32 Refl, Ascii 106 Refl, Ascii 107 Refl,
  Ascii 108 Refl]))))) []) []
```

3.11. Canonical S-Expr Transport

3.11.1. Analysis

Section 6.1 of the original s-expr document states:

```
NOTE: '\\' line wrapping per RFC 8792

This canonical representation is used for digital signature \
\purposes,
transmission, etc.  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, to be reasonably economical, \
\and to
be unique for any S-expression.

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.
```

3.11.2. Formalization

The canonical transport is actually a profile of the advanced transport, so we can reuse our previous types:

First we declare an abstract type for the canonical s-expr, as it is an inductive type:

```
data CanonicalSExpr : List Bits8 -> Type
```

Then a type for a list of canonical s-expr:

```
data CanonicalSExprList : List Bits8 -> Type where
  Nil : CanonicalSExprList []
  (::) : CanonicalSExpr xs -> CanonicalSExprList ys ->
    CanonicalSExprList (xs ++ ys)
```

And finally our concrete type for a canonical s-expr:

```
data CanonicalSExpr : List Bits8 -> Type where
  MkCanonical : Verbatim xs -> CanonicalSExpr xs
  MkCanonicalHint : Verbatim xs -> Verbatim ys ->
    CanonicalSExpr (91 :: xs ++ [93] ++ ys)
  MkCanonicalList : CanonicalSExprList xs ->
    CanonicalSExpr (40 :: xs ++ [41])
```

3.11.3. Validation

Here we prove that all the examples in section 6.1 of the original document are valid instances of the Canonical type:

```
NOTE: '\\' line wrapping per RFC 8792

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))
```

* (6:issuer3:bob)

```
testCanonical1 : CanonicalSExpr [40, 54, 58, 105, 115, 115,
  117, 101, 114, 51, 58, 98, 111, 98, 41]
testCanonical1 = MkCanonicalList [MkCanonical
  (MkVerbatim [105, 115, 115, 117, 101, 114]),
  MkCanonical (MkVerbatim [98, 111, 98])]
```

* (4:icon[12:image/bitmap]9:xxxxxxxxx)

```
testCanonical2 : CanonicalSExpr [40, 52, 58, 105, 99, 111, 110,
  91, 49, 50, 58, 105, 109, 97, 103, 101, 47, 98, 105, 116,
  109, 97, 112, 93, 57, 58, 120, 120, 120, 120, 120, 120,
  120, 120, 120, 41]
testCanonical2 = MkCanonicalList [MkCanonical
  (MkVerbatim [105, 99, 111, 110]), MkCanonicalHint
  (MkVerbatim [105, 109, 97, 103, 101, 47, 98, 105, 116,
  109, 97, 112]) (MkVerbatim [120, 120, 120, 120, 120, 120,
  120, 120, 120])]
```

* (7:subject(3:ref5:alice6:mother))

```
testCanonical3 : CanonicalSExpr [40, 55, 58, 115, 117, 98, 106,
  101, 99, 116, 40, 51, 58, 114, 101, 102, 53, 58, 97, 108,
  105, 99, 101, 54, 58, 109, 111, 116, 104, 101, 114, 41, 41]
testCanonical3 = MkCanonicalList [MkCanonical
  (MkVerbatim [115, 117, 98, 106, 101, 99, 116]),
  MkCanonicalList [MkCanonical (MkVerbatim [114, 101, 102]),
  MkCanonical (MkVerbatim [97, 108, 105, 99, 101]),
  MkCanonical (MkVerbatim [109, 111, 116, 104, 101, 114])]]
```

3.12. Basic S-Expr Transport

3.12.1. Analysis

Section 6.2 of the original s-expr document states:

```
NOTE: '\\' line wrapping per RFC 8792

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

        -- the canonical representation

        -- an rfc-2045 base-64 representation of the canonical \
\representation,
           surrounded by braces.

The transport mechanism is intended to provide a universal means \
\of
representing S-expressions for transport from one machine to \
\another.
```

There is no possible BNF that is sound for a base 64 representation of an underlying s-expr.

3.12.2. Formalization

The basic transport is also a profile of the advanced transport, so we can reuse some previous types:

We first redefine Base64Full without white spaces:

```
data BasicBase64Full : List Bits8 -> Type where
  MkBasicBase64Full : (x1 : Bits8) -> (x2 : Bits8) ->
    (x3 : Bits8) ->
    BasicBase64Full (b641 x1 ++ b642 x1 x2 ++ b643 x2 x3 ++
      b644 x3s)
```

Then a list of these:

```
namespace BasicBase64
  public export
  data BasicBase64List : List Bits8 -> Type where
    Nil : BasicBase64List []
    (::) : BasicBase64Full xs -> BasicBase64List ys ->
      BasicBase64List (xs ++ ys)
```

And a type for a base 64 encoding for lengths that are not a multiple of 3:

```
data BasicBase64End : List Bits8 -> Type where
  BasicEndOnePadPad : (x1 : Bits8) ->
    BasicBase64End (b641 x1 ++ b642 x1 0 ++ [61, 61])
  BasicEndOnePad : (x1 : Bits8) ->
    BasicBase64End (b641 x1 ++ b642 x1 0 ++ [61])
  BasicEndOne : (x1 : Bits8) ->
    BasicBase64End (b641 x1 ++ b642 x1 0)
  BasicEndTwoPad : (x1 : Bits8) -> (x2 : Bits8) ->
    BasicBase64End (b641 x1 ++ b642 x1 x2 ++ b643 x2 0 ++ [61])
  BasicEndTwo : (x1 : Bits8) -> (x2 : Bits8) ->
    BasicBase64End (b641 x1 ++ b642 x1 x2 ++ b643 x2 0)
```

And a basic base 64 type:

```
data BasicBase64 : List Bits8 -> Type where
  BasicBase64Mult3 : BasicBase64List xs -> BasicBase64 xs
  BasicBase64Non : BasicBase64List xs -> BasicBase64End ys ->
    BasicBase64 (xs ++ ys)
```

Then we need to define three base64 encoding functions, one for each variant:

```
base64 : List Bits8 -> List Bits8
base64 [] = []
base64 [x1] = b641 x1 ++ b642 x1 0 ++ [61, 61]
base64 [x1, x2] = b641 x1 ++ b642 x1 x2 ++ b643 x2 0 ++ [61]
base64 (x1 :: x2 :: x3 :: xs) = b641 x1 ++ b642 x1 x2 ++
  b643 x2 x3 ++ b644 x3 ++ base64 xs

base64OnePad : List Bits8 -> List Bits8
base64OnePad [] = []
base64OnePad [x1] = b641 x1 ++ b642 x1 0 ++ [61]
base64OnePad [x1, x2] = b641 x1 ++ b642 x1 x2 ++ b643 x2 0
base64OnePad (x1 :: x2 :: x3 :: xs) = b641 x1 ++ b642 x1 x2 ++
  b643 x2 x3 ++ b644 x3 ++ base64OnePad xs

base64NoPad : List Bits8 -> List Bits8
base64NoPad [] = []
base64NoPad [x1] = b641 x1 ++ b642 x1 0
base64NoPad [x1, x2] = b641 x1 ++ b642 x1 x2 ++ b643 x2 0
base64NoPad (x1 :: x2 :: x3 :: xs) = b641 x1 ++ b642 x1 x2 ++
  b643 x2 x3 ++ b644 x3 ++ base64NoPad xs
```

And finally our type for a brace notation for base 64:

```
data BasicSExpr : List Bits8 -> Type where
  MkBasicCanonical : CanonicalSExpr xs -> BasicSExpr xs
  MkBasicBase64 : CanonicalSExpr xs -> BasicBase64 ys ->
   (prf : (base64 xs == ys) === True) ->
   BasicSExpr (123 :: ys ++ [123])
  MkBasicBase64OnePad : CanonicalSExpr xs -> BasicBase64 ys ->
   (prf : (base64OnePad xs == ys) === True) ->
   BasicSExpr (123 :: ys ++ [123])
  MkBasicBase64NoPad : CanonicalSExpr xs -> BasicBase64 ys ->
    (prf : (base64NoPad xs == ys) === True) ->
    BasicSExpr (123 :: ys ++ [123])
```

3.12.3. Validation

Here we prove that the first example in section 6.2 of the original document is a valid instance of the Basic type:

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

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

        {KDE6YTE6YjE6YykA}

                (this is the same S-expression encoded in base-64)
```

* (1:a1:b1:c)

```
testBasic1 : BasicSExpr [40, 49, 58, 97, 49, 58, 98, 49, 58,
  99, 41]
testBasic1 = MkBasicCanonical (MkCanonicalList [MkCanonical
  (MkVerbatim [97]), MkCanonical (MkVerbatim [98]), MkCanonical
  (MkVerbatim [99])])
```

3.13. Array-Layout

3.13.1. Analysis

Section 8.2 of the original s-expr document states:

> NOTE: '\\' line wrapping per RFC 8792
>
> Here each S-expression is represented as a contiguous array of \
> \bytes.
> The first byte 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-byte \
> \integer
> indicating the size (in bytes) of the following representation.\
> \ Here
> k is an integer that depends on the implementation, it might be
> anywhere from 2 to 8, but 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 length 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.

The endianness of the length field is not specified, so we assume that both little and big endianness can be used.

Furthermore section 8.2.1 states:

> This is represented as follows:
>
> ```
> 01 <length> <octet-string>
> ```

Section 8.2.2 states:

> This is represented as follows:
>
> ```
> 02 <length>
>   01 <length> <octet-string>    /* for display-type */
>   01 <length> <octet-string>    /* for octet-string */
> ```

And section 8.2.3 states:

> This is represented as
>
> ```
> 03 <length> <item1> <item2> <item3> ... <itemn> 00
> ```

3.13.2. Formalization

First we define a type for the endianness of the length:

```
data Endianness = Big | Little
```

Then a function that converts a natural number into a memory representation of a specified endianness and length:

```
convert' : Nat -> Nat -> List Bits8
convert' 0 _ = []
convert' (S k) n =
  let (d, m) = divmodNatNZ n 256 ItIsSucc
  in cast m :: convert' k d

export
convert : Endianness -> Nat -> Nat -> List Bits8
convert Big k j = convert' k j
convert Little k j = reverse (convert' k j)
```

Then we define a type for the array representation of an octet-string:

```
      data ArrayOctetString :
        Endianness -> Nat -> List Bits8 -> Type where
        MkArrayOctetString : (xs : List Bits8) ->
         ArrayOctetString e l (1 :: convert e l (length xs) ++ xs)
```

Then for an octet-string with display-hint:

```
      data ArrayWithHint : Endianness -> Nat -> List Bits8 ->
        Type where
        MkArrayWithHint : ArrayOctetString e l xs ->
          ArrayOctetString e l ys ->
          ArrayWithHint e l (2 :: convert e l (length xs +
          length ys) ++ xs ++ ys)
```

As usual an abstract type for an inductive type:

```
      data ArraySExpr : Endianness -> Nat -> List Bits8 -> Type
```

Then a list of memory array:

```
      namespace Array
        public export
        data ArrayList : Endianness -> Nat -> List Bits8 ->
          Type where
          Nil : ArrayList e l []
          (::) : ArraySExpr e l xs -> ArrayList e l ys ->
            ArrayList e l (xs ++ ys)
```

And finally the array memory type:

```
      data ArraySExpr : Endianness -> Nat -> List Bits8 -> Type where
        ArraySExprOctetString : ArrayOctetString e l xs ->
          ArraySExpr e l xs
        ArraySExprWithHint : ArrayWithHint e l xs ->
          ArraySExpr e l xs
        ArraySExprList : ArrayList e l xs ->
          ArraySExpr e l (3 :: convert e l (1 + length xs) ++
          xs ++ [0])
```

3.13.3. Verification

Here we prove that all the examples in section 8.2 of the original document are valid instances of the ArraySExpr type:

* abc

For example (here k = 2)

```
        01 0003 a b c

testArray1 : ArraySExpr Little 2 [1, 0, 3, 97, 98, 99]
testArray1 = ArraySExprOctetString
  (MkArrayOctetString [97, 98, 99])
```

* [gif] #61626364#

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

testArray2 : ArraySExpr Little 2 [2, 0, 13, 1, 0, 3, 103,
  105, 102, 1, 0, 4, 97, 98, 99, 100]
testArray2 = ArraySExprWithHint (MkArrayWithHint
  (MkArrayOctetString [103, 105, 102])
  (MkArrayOctetString [97, 98, 99, 100]))
```

* (abc [d]ef (g))

NOTE: '\\' line wrapping per RFC 8792

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
```

```
testArray3 : ArraySExpr Little 2 [3, 0, 27, 1, 0, 3, 97, 98,
  99, 2, 0, 9, 1, 0, 1, 100, 1, 0, 2, 101, 102, 3, 0, 5, 1,
  0, 1, 103, 0, 0]
testArray3 = ArraySExprList [ArraySExprOctetString
  (MkArrayOctetString [97, 98, 99]), ArraySExprWithHint
  (MkArrayWithHint (MkArrayOctetString [100])
  (MkArrayOctetString [101, 102])), ArraySExprList
  [ArraySExprOctetString (MkArrayOctetString [103])]]
```

4. Informative References

[ComputerateSpecification]
Petit-Huguenin, M., "Computerate Specification", Work in Progress, Internet-Draft, draft-petithuguenin-computerate-specification, 3 February 2024, <https://datatracker.ietf.org/doc/draft-petithuguenin-computerate-specification>.

[Idris2] "Documentation for the Idris 2 Language — Idris2 0.0 documentation", Accessed 31 January 2023, <https://idris2.readthedocs.io/en/latest/>.

[RFC8792] Watsen, K., Auerswald, E., Farrel, A., and Q. Wu, "Handling Long Lines in Content of Internet-Drafts and RFCs", RFC 8792, DOI 10.17487/RFC8792, June 2020, <https://www.rfc-editor.org/info/rfc8792>.

[SPKI-SExpr]
Rivest, R. L. and D. E. Eastlake 3rd, "SPKI S-Expressions", Work in Progress, Internet-Draft, draft-rivest-sexp, 16 April 2024, <https://datatracker.ietf.org/doc/draft-rivest-sexp>.

Appendix A. Code Extraction and Verification

To verify that the proofs in this document are correct, the first step is to install [Idris2].

Then the various Idris2 fragments in this document can be extracted as a complete file by running the following command:

```
xmllint --noent --nocdata \
--xpath "//sourcecode[@name='FormalSexpr.idr']/text()" \
draft-petithuguenin-ufmrg-formal-sexpr-06.xml \
| sed "s/&lt;/</g; s/&gt;/>/g; s/amp;//g" >FormalSexpr.idr
```

And finally the proofs can be validated by using the following command

```
idris2 -q -c FormalSexpr.idr
```

Acknowledgements

Thanks to Erik Auerswald and Stephane Bryant for the comments, suggestions and questions that helped improve this document.

No technology that cannot explain its own results (LLM, AI/ML) have been involved in the creation of this document.

Changelog

Since draft-petithuguenin-ufmrg-formal-sexpr-05:

- code is now a named module so it can be exported in a package
- redefine the show function
- replace the deprecated SIsNonZero function

Since draft-petithuguenin-ufmrg-formal-sexpr-04:

- default encoding reverted to "text/plain; charset=iso-8859-1"
- more nits and clarifications

Since draft-petithuguenin-ufmrg-formal-sexpr-03:

- more nits and clarifications
- add proof of equivalence using symmetry
- change default display-hint to application/octet-stream

Since draft-petithuguenin-ufmrg-formal-sexpr-02:

- fix another instance of incorrect rendering
- reformat some example for clarity
- add "Equality of Octet-String" section
- nits and some clarification

Since draft-petithuguenin-ufmrg-formal-sexpr-01:

- incorrect rendering of a string with #
- add RFC 8792 headers
- remove a comment about tokens that was incorrect
- add REPL example to display octet-strings

Since draft-petithuguenin-ufmrg-formal-sexpr-00:

- add forgotten proofs for testList2 and testList3
- some editing for clarity
- many nit fixes

Author’s Address

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