

Ethereum's Recursive Length Prefix in ACL2

Alessandro Coglio





Ethereum is a major public blockchain with smart contracts and a cryptocurrency.



Ethereum uses Recursive Length Prefix (RLP) to encode a variety of data structures, including transactions and blocks.



This work is a development, in ACL2, of a formal specification of RLP encoding and a verified implementation of RLP decoding.

RLP encodes nested byte sequences into flat byte sequences.

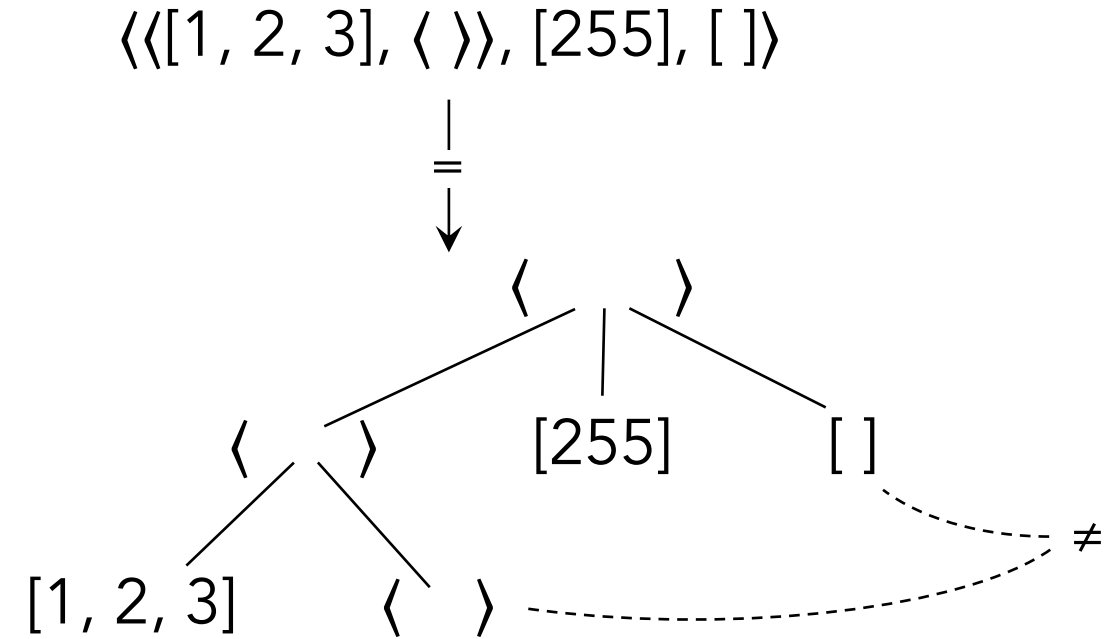
A nested byte sequence...

$\langle\langle[1, 2, 3], \langle \rangle\rangle, [255], []\rangle$

RLP encodes nested byte sequences into flat byte sequences.

A nested byte sequence...

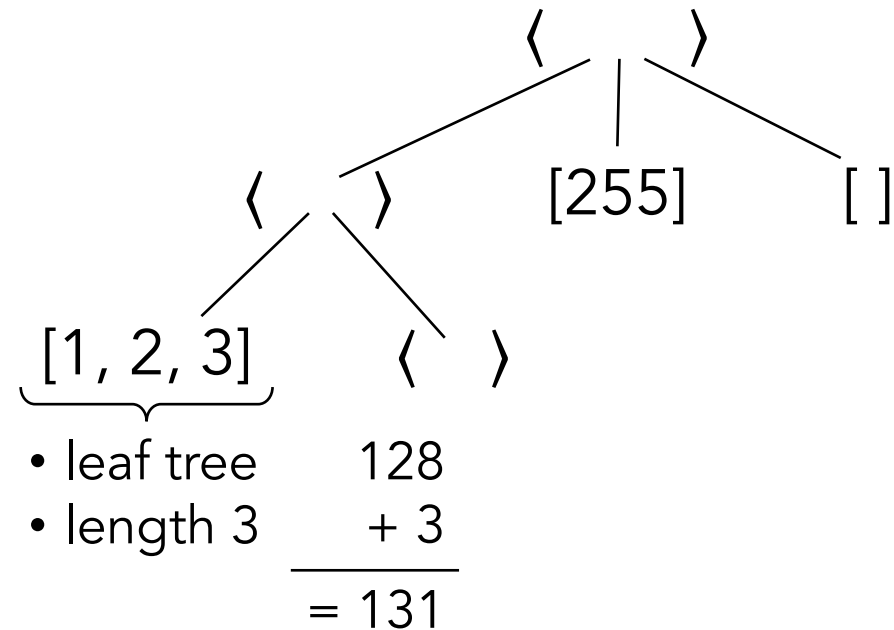
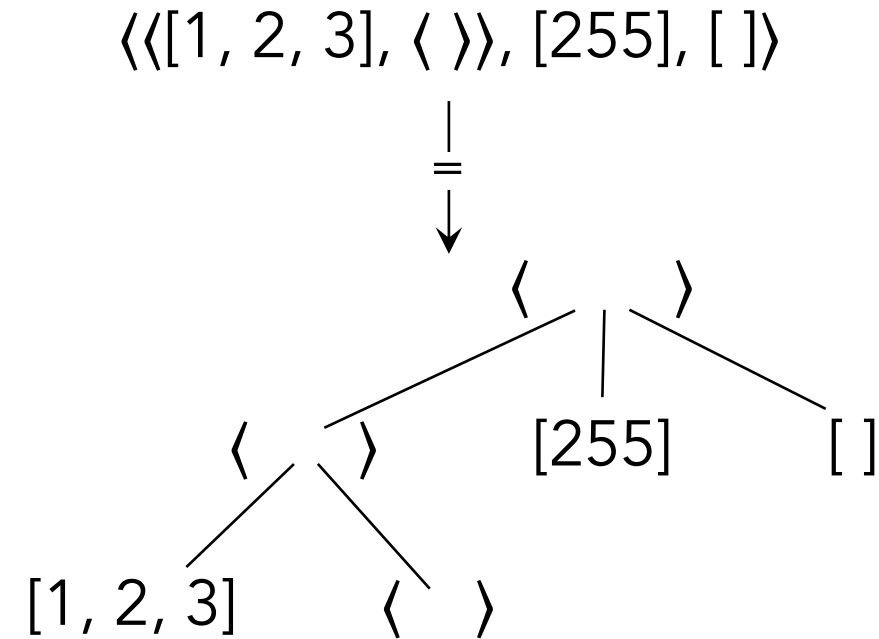
... i.e. a finitely branching ordered tree
with flat byte sequences at leaf nodes
and no extra info at branching nodes...



A nested byte sequence...

... i.e. a finitely branching ordered tree
with flat byte sequences at leaf nodes
and no extra info at branching nodes...

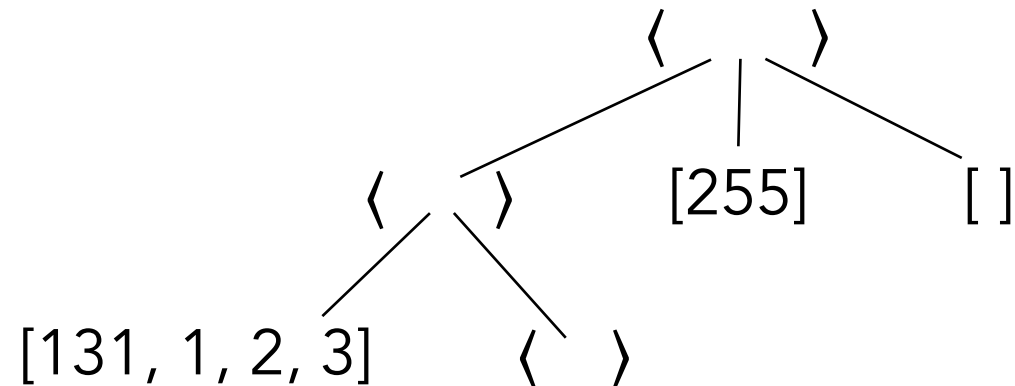
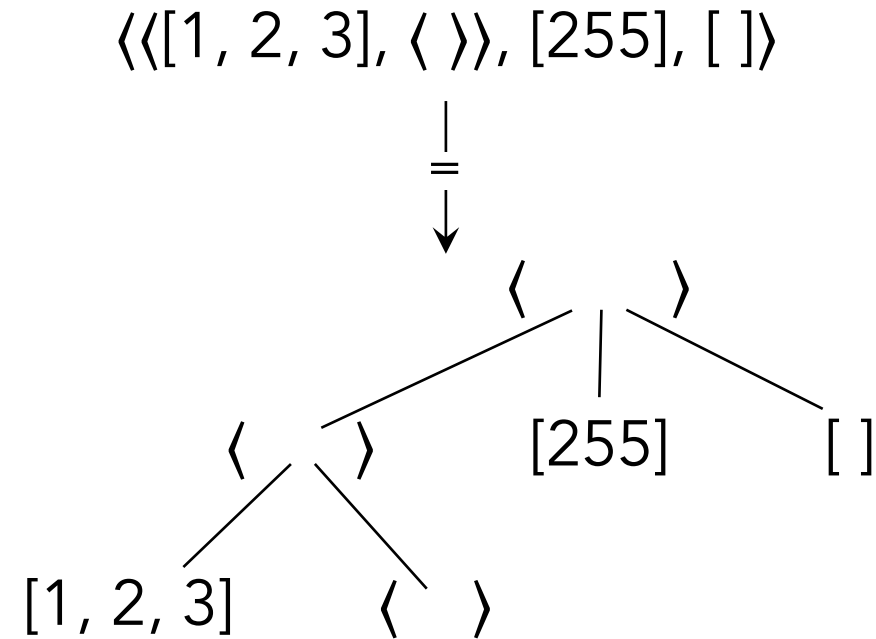
... is encoded by prepending nodes with a few bytes that describe the node kind (leaf vs. branching) and the number of subsequent bytes.



A nested byte sequence...

... i.e. a finitely branching ordered tree
with flat byte sequences at leaf nodes
and no extra info at branching nodes...

... is encoded by prepending nodes with a few bytes that describe the node kind (leaf vs. branching) and the number of subsequent bytes.

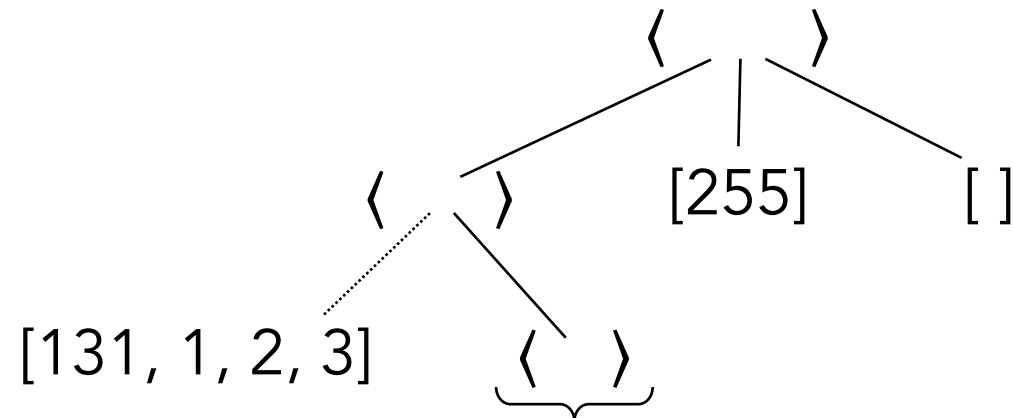
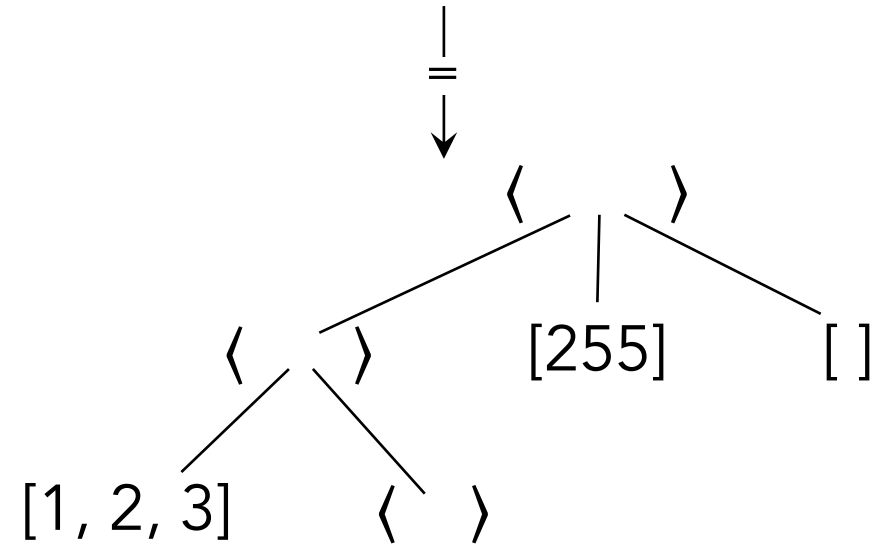


A nested byte sequence...

... i.e. a finitely branching ordered tree
with flat byte sequences at leaf nodes
and no extra info at branching nodes...

... is encoded by prepending nodes
with a few bytes that describe
the node kind (leaf vs. branching)
and the number of subsequent bytes.

$\langle \langle [1, 2, 3], \langle \rangle \rangle, [255], [] \rangle$

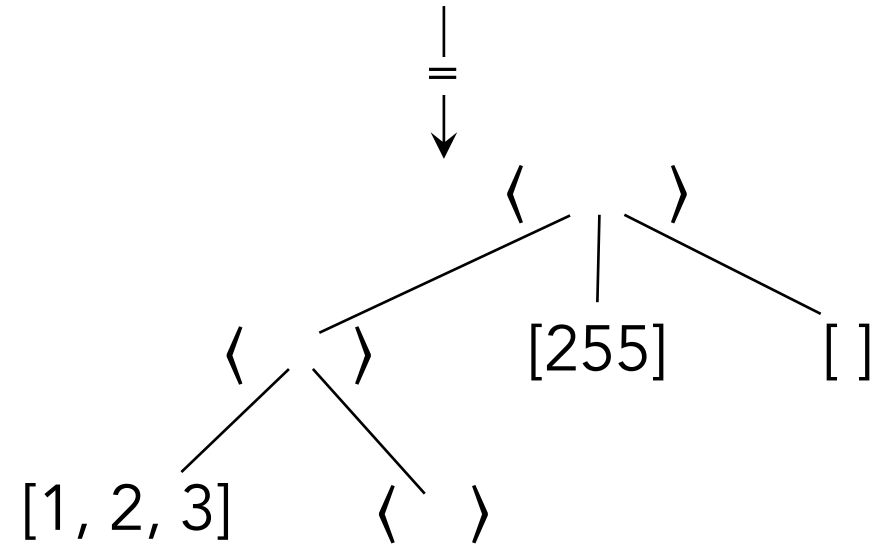


- branching tree 192
 - subtree length 0 + 0
-
- = 192

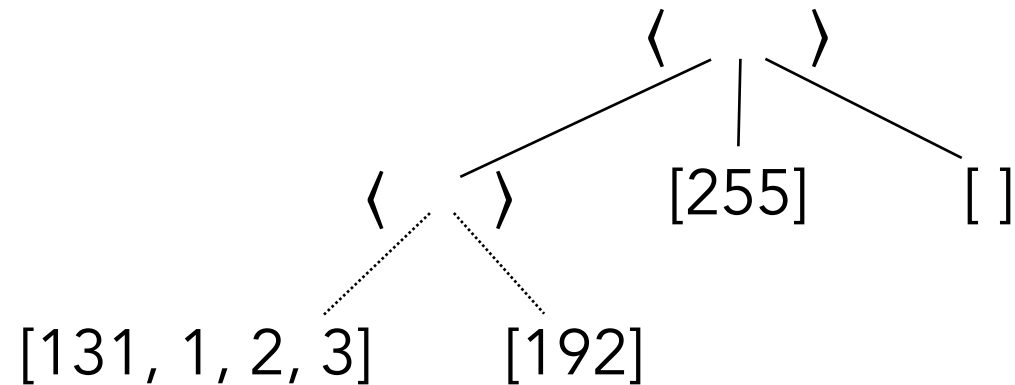
A nested byte sequence...

... i.e. a finitely branching ordered tree
with flat byte sequences at leaf nodes
and no extra info at branching nodes...

$\langle \langle [1, 2, 3], \langle \rangle \rangle, [255], [] \rangle$



... is encoded by prepending nodes
with a few bytes that describe
the node kind (leaf vs. branching)
and the number of subsequent bytes.

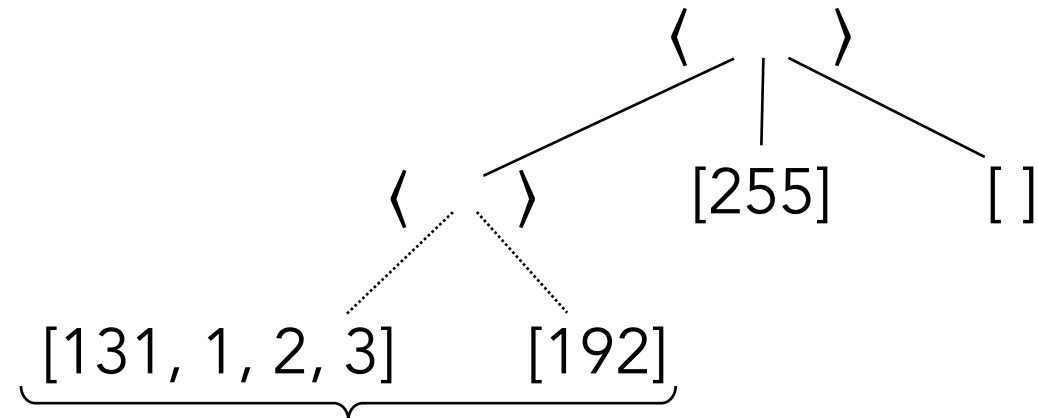
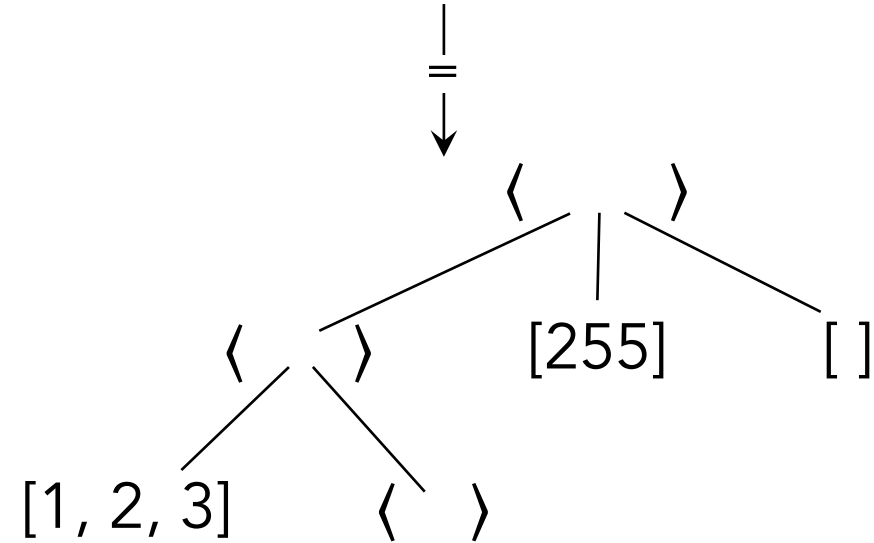


A nested byte sequence...

... i.e. a finitely branching ordered tree with flat byte sequences at leaf nodes and no extra info at branching nodes...

... is encoded by prepending nodes with a few bytes that describe the node kind (leaf vs. branching) and the number of subsequent bytes.

$\langle \langle [1, 2, 3], \langle \rangle \rangle, [255], [] \rangle$



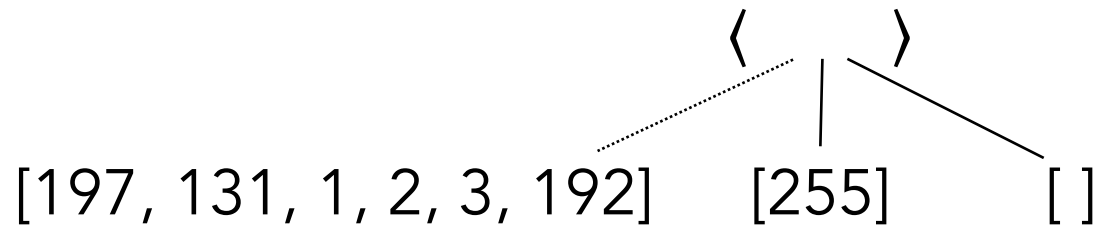
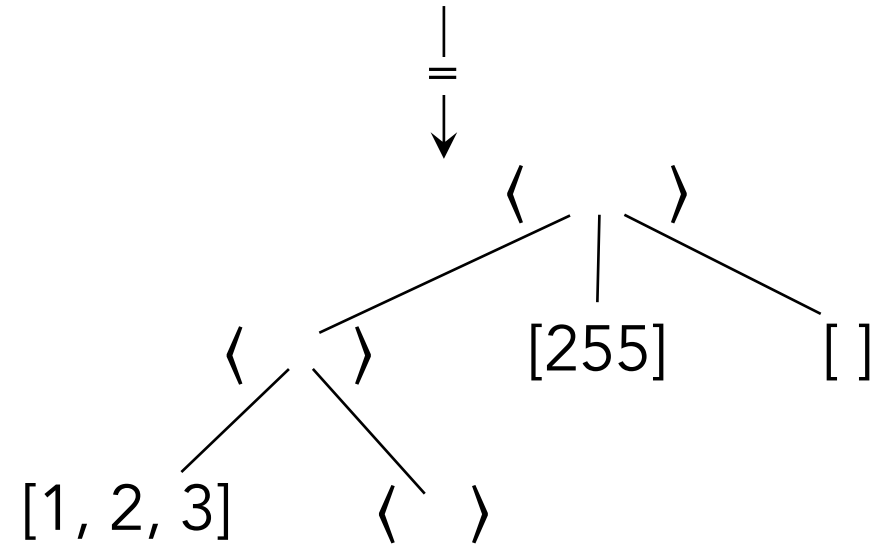
• branching tree	192
• subtree length 5	+ 5
	<hr/>
	= 197

A nested byte sequence...

... i.e. a finitely branching ordered tree with flat byte sequences at leaf nodes and no extra info at branching nodes...

... is encoded by prepending nodes with a few bytes that describe the node kind (leaf vs. branching) and the number of subsequent bytes.

$\langle \langle [1, 2, 3], \langle \rangle \rangle, [255], [] \rangle$

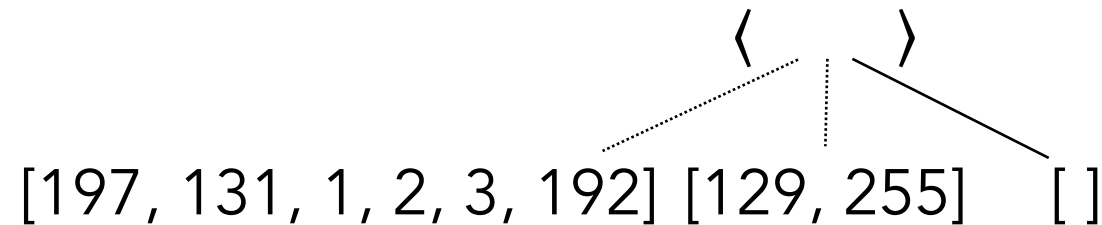
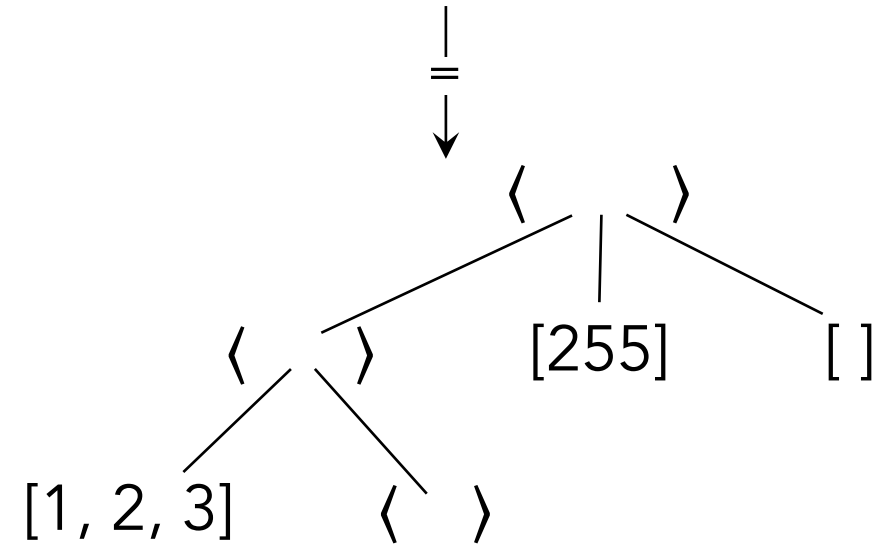


A nested byte sequence...

... i.e. a finitely branching ordered tree with flat byte sequences at leaf nodes and no extra info at branching nodes...

... is encoded by prepending nodes with a few bytes that describe the node kind (leaf vs. branching) and the number of subsequent bytes.

$\langle\langle[1, 2, 3], \langle \rangle\rangle, [255], []\rangle$

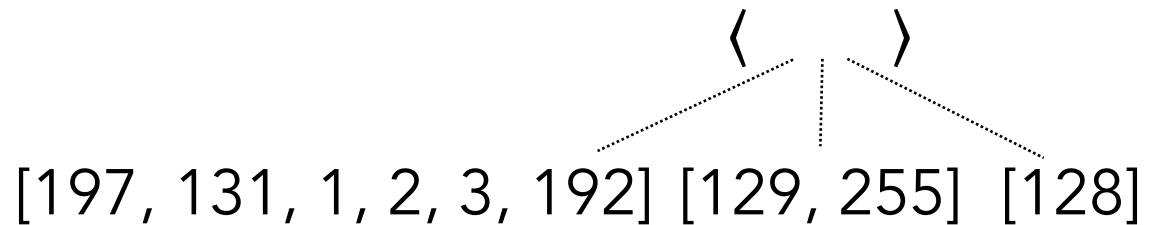
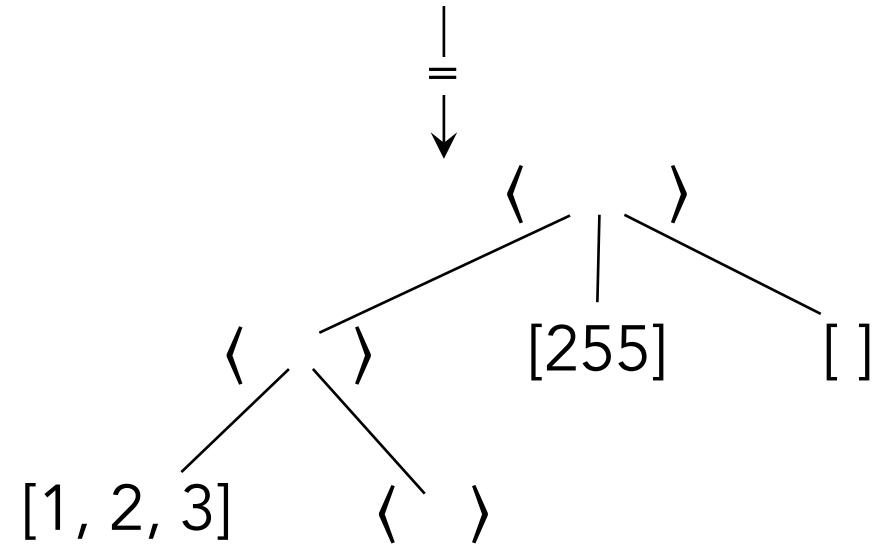


A nested byte sequence...

... i.e. a finitely branching ordered tree
with flat byte sequences at leaf nodes
and no extra info at branching nodes...

... is encoded by prepending nodes
with a few bytes that describe
the node kind (leaf vs. branching)
and the number of subsequent bytes.

$\langle \langle [1, 2, 3], \langle \rangle \rangle, [255], [] \rangle$

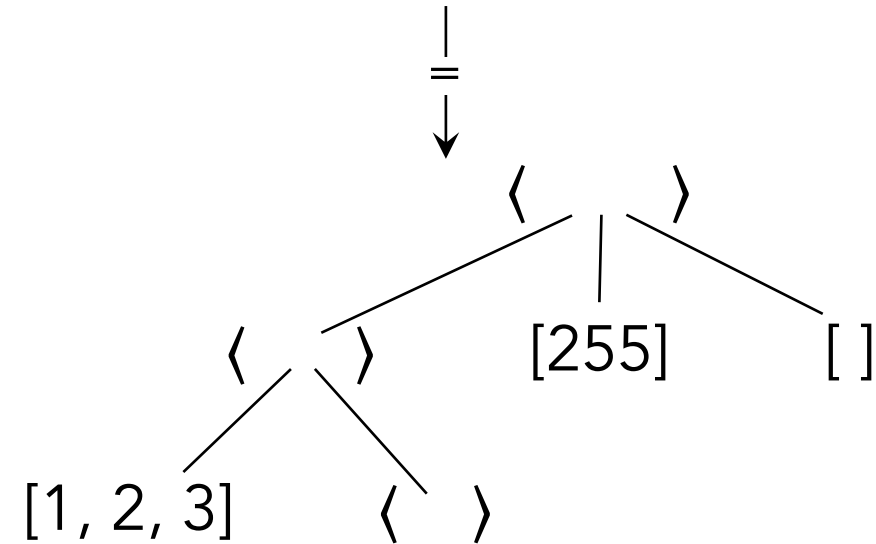


A nested byte sequence...

... i.e. a finitely branching ordered tree
with flat byte sequences at leaf nodes
and no extra info at branching nodes...

... is encoded by prepending nodes
with a few bytes that describe
the node kind (leaf vs. branching)
and the number of subsequent bytes.

$\langle \langle [1, 2, 3], \langle \rangle \rangle, [255], [] \rangle$



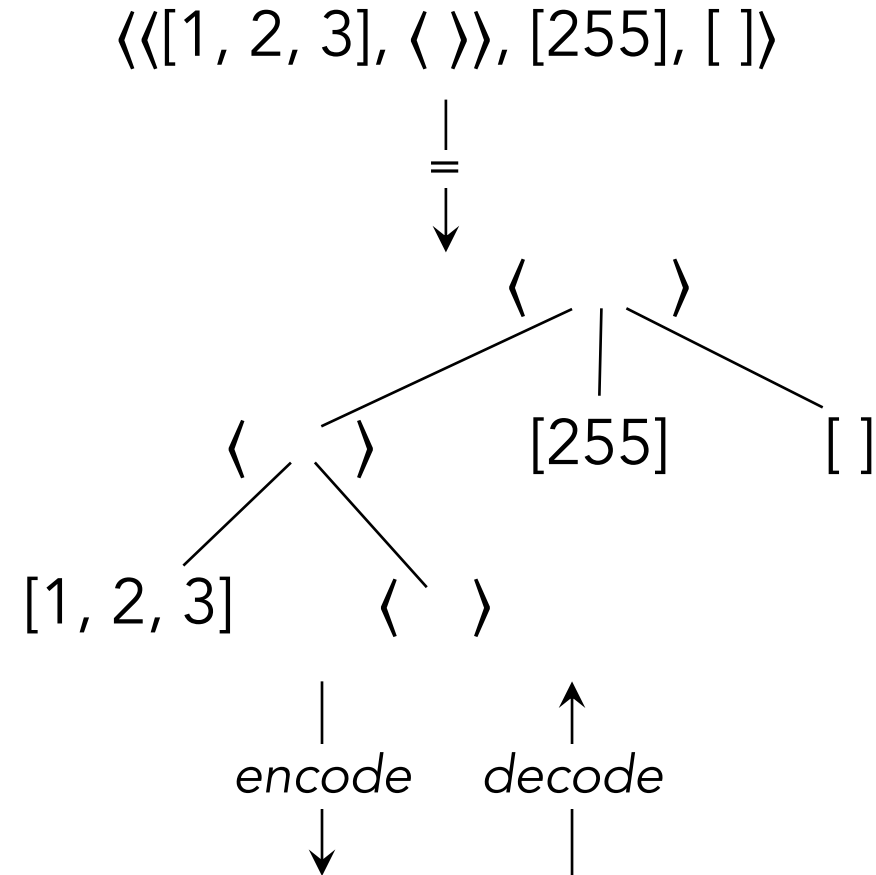
[201, 197, 131, 1, 2, 3, 192, 129, 255, 128]

RLP encodes nested byte sequences into flat byte sequences.

A nested byte sequence...


... i.e. a finitely branching ordered tree with flat byte sequences at leaf nodes and no extra info at branching nodes...


... is encoded by prepending nodes with a few bytes that describe the node kind (leaf vs. branching) and the number of subsequent bytes.



$[201, 197, 131, 1, 2, 3, 192, 129, 255, 128]$

RLP is described in the Ethereum Wiki, using Python code.

 Why GitHub? ▾ Team Enterprise Explore ▾ Marketplace Pricing ▾

 [ethereum](#) / [wiki](#) 1.3k 13.2k 2.3k

[Code](#) [Issues 18](#) [Pull requests 3](#) [Actions](#) [Projects 0](#) [Wiki](#) [Security 0](#) [Insights](#)

RLP

Chris Chinchilla edited this page on Aug 2, 2019 · 36 revisions

Contents

- [Definition](#)
- [Examples](#)
- [RLP decoding](#)
- [Implementations](#)

The purpose of RLP (Recursive Length Prefix) is to encode arbitrarily nested arrays of binary data, and RLP is the main encoding method used to serialize objects in Ethereum. The only purpose of RLP is to encode structure; encoding specific data types (eg. strings, floats) is left up to higher-order protocols; but positive RLP integers must be represented in big endian binary form with no leading zeroes (thus making the integer value zero be equivalent to the empty byte array). Deserialised positive integers with leading zeroes must be treated as invalid. The integer representation of string length must also be encoded this way, as well as integers in the payload. Additional information can be found in the Ethereum yellow paper Appendix B.

If one wishes to use RLP to encode a dictionary, the two suggested canonical forms are to either use `[[k1,v1],[k2,v2]]...` with keys in lexicographic order or to use the higher-level [Patricia Tree](#) encoding as Ethereum does.

In summary, RLP is like a binary encoding of JSON, if JSON were restricted only to strings and arrays.

Definition

The RLP encoding function takes in an item. An item is defined as follows:

[Pages 214](#)

Basics

- [Home](#)
- [Ethereum Whitepaper](#)
- [Ethereum Introduction](#)
- [Uses: DAOs and dapps](#)
- [Getting Ether](#)
- [Releases](#)
- [FAQs](#)
- [Design Rationale](#)
- EVM intro: [Ethereum Yellow Paper](#), [Beige Paper](#) and [Py-EVM](#).
- [Wiki for \(old\) website](#) (still a good introduction)
- [Glossary](#)

R&D

- [Sharding introduction & R&D Compendium, FAQs & roadmap](#)
- [Casper Proof-of-Stake compendium and FAQs.](#)

RLP is described in the Ethereum Wiki, using Python code.

a leaf tree $[x]$ with $x < 128$
is encoded as itself, i.e. $[x]$

a leaf tree $[x_1, \dots, x_n]$ with $n < 56$
is encoded as $[128+n, x_1, \dots, x_n]$

a leaf tree $[x_1, \dots, x_n]$ with $n < 2^{64}$
is encoded as $[183+m, y_1, \dots, y_m, x_1, \dots, x_n]$
where $[y_1, \dots, y_m]$ is n in big endian base 256

a branch tree is encoded by
concatenating the subtree encodings
into $[x_1, \dots, x_n]$ and prepending with
either $[192+n]$ when $n < 56$,
or $[247+m, y_1, \dots, y_m]$ when $n < 2^{64}$
where $[y_1, \dots, y_m]$ is n in big endian base 256

- For a single byte whose value is in the `[0x00, 0x7f]` range, that byte is its own RLP encoding.
- Otherwise, if a string is 0-55 bytes long, the RLP encoding consists of a single byte with value `0x80` plus the length of the string followed by the string. The range of the first byte is thus `[0x80, 0xb7]`.
- If a string is more than 55 bytes long, the RLP encoding consists of a single byte with value `0xb7` plus the length in bytes of the length of the string in binary form, followed by the length of the string, followed by the string. For example, a length-1024 string would be encoded as `\xb9\x04\x00` followed by the string. The range of the first byte is thus `[0xb8, 0xbf]`.
- If the total payload of a list (i.e. the combined length of all its items being RLP encoded) is 0-55 bytes long, the RLP encoding consists of a single byte with value `0xc0` plus the length of the list followed by the concatenation of the RLP encodings of the items. The range of the first byte is thus `[0xc0, 0xf7]`.
- If the total payload of a list is more than 55 bytes long, the RLP encoding consists of a single byte with value `0xf7` plus the length in bytes of the length of the payload in binary form, followed by the length of the payload, followed by the concatenation of the RLP encodings of the items. The range of the first byte is thus `[0xf8, 0xff]`.

In code, this is:

```
def rlp_encode(input):
    if isinstance(input, str):
        if len(input) == 1 and ord(input) < 0x80: return input
        else: return encode_length(len(input), 0x80) + input
    elif isinstance(input, list):
        output = ''
        for item in input: output += rlp_encode(item)
        return encode_length(len(output), 0xc0) + output

def encode_length(L, offset):
    if L < 56:
        return chr(L + offset)
    elif L < 256*8:
        BL = to_binary(L)
        return chr(len(BL) + offset + 55) + BL
    else:
        raise Exception("input too long")

def to_binary(x):
    if x == 0:
        return ''
    else:
        return to_binary(int(x / 256)) + chr(x % 256)
```

[tools, wallets, dapp
browsers and other
projects](#)

[DApp Development](#)

[Infrastructure](#)

- [Chain Spec Format](#)
- [Inter-exchange Client
Address Protocol](#)
- [URL Hint Protocol](#)
- [NatSpec Determination](#)
- [Network Status](#)
- [Raspberry Pi](#)
- [Mining](#)
- [Licensing](#)
- [Consortium Chain
Development](#)

[Networking](#)

- [Ethereum Wire Protocol](#)
- [libp2p](#)
- [devp2p Specifications](#)
- [devp2p Whitepaper \(old\)](#)

[Ethereum
Technologies](#)

- [RLP Encoding](#)
- [Patricia Tree](#)
- [Web3 Secret Storage](#)
- [Light client protocol](#)
- [Subleties](#)
- [Solidity Documentation](#)
- [NatSpec Format](#)
- [Contract ABI](#)
- [Bad Block Reporting](#)
- [Bad Chain Canary](#)

RLP is described in the Ethereum Wiki, using Python code.

an encoding is decoded by
“following the instructions”
in the first (few) byte(s),
recursively decoding subtrees

decoding is
more complicated
than encoding

the Python code
had an error,
fixed as a result
of this ACL2 work

```
def rlp_decode(input):
    if len(input) == 0:
        return
    output = ''
    (offset, dataLen, type) = decode_length(input)
    if type is str:
        output = instantiate_str(substr(input, offset, dataLen))
    elif type is list:
        output = instantiate_list(substr(input, offset, dataLen))
    output = output + rlp_decode(substr(input, offset + dataLen))
    return output

def decode_length(input):
    length = len(input)
    if length == 0:
        raise Exception("input is null")
    prefix = ord(input[0])
    if prefix <= 0x7f:
        return (0, 1, str)
    elif prefix <= 0xb7 and length > prefix - 0x80:
        strLen = prefix - 0x80
        if strLen == 1 and ord(input[1]) <= 0x7f:
            raise Exception("single byte below 128 must be encoded as itself")
        return (1, strLen, str)
    elif prefix <= 0xbf and length > prefix - 0xb7 and length > prefix - 0xb7 + to_integer(substr(input, 1, prefix - 0xb7)):
        lenOfStrLen = prefix - 0xb7
        if input[1] == 0:
            raise Exception("multi-byte length must have no leading zero");
        strLen = to_integer(substr(input, 1, lenOfStrLen))
        if strLen < 56:
            raise Exception("length below 56 must be encoded in one byte");
        return (1 + lenOfStrLen, strLen, str)
    elif prefix <= 0xf7 and length > prefix - 0xc0:
        listLen = prefix - 0xc0;
        return (1, listLen, list)
    elif prefix <= 0xff and length > prefix - 0xf7 and length > prefix - 0xf7 + to_integer(substr(input, 1, prefix - 0xf7)):
        lenOfListLen = prefix - 0xf7
        if input[1] == 0:
            raise Exception("multi-byte length must have no leading zero");
        listLen = to_integer(substr(input, 1, lenOfListLen))
        if listLen < 56:
            raise Exception("length below 56 must be encoded in one byte");
        return (1 + lenOfListLen, listLen, list)
    else:
        raise Exception("input don't conform RLP encoding form")

def to_integer(b):
```

RLP is described in the Ethereum Yellow Paper, formally.

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER

BYZANTIUM VERSION 7e819ec - 2019-10-20

DR. GAVIN WOOD
FOUNDER, ETHEREUM & PARITY
GAVIN@PARITY.IO

ABSTRACT. The blockchain paradigm when coupled with cryptographically-secured transactions has demonstrated its utility through a number of projects, with Bitcoin being one of the most notable ones. Each such project can be seen as a simple application on a decentralised, but singleton, compute resource. We can call this paradigm a transactional singleton machine with shared-state.

Ethereum implements this paradigm in a generalised manner. Furthermore it provides a plurality of such resources, each with a distinct state and operating code but able to interact through a message-passing framework with others. We discuss its design, implementation issues, the opportunities it provides and the future hurdles we envisage.

1. INTRODUCTION

With ubiquitous internet connections in most places of the world, global information transmission has become incredibly cheap. Technology-rooted movements like Bitcoin have demonstrated through the power of the default, consensus mechanisms, and voluntary respect of the social contract, that it is possible to use the internet to make a decentralised value-transfer system that can be shared across the world and virtually free to use. This system can be said to be a very specialised version of a cryptographically secure, transaction-based state machine. Follow-up

is often lacking, and plain old prejudices are difficult to shake.

Overall, we wish to provide a system such that users can be guaranteed that no matter with which other individuals, systems or organisations they interact, they can do so with absolute confidence in the possible outcomes and how those outcomes might come about.

1.2. Previous Work. Buterin [2013a] first proposed the kernel of this work in late November, 2013. Though now evolved in many ways, the key functionality of a blockchain with a Turing complete language and an effectively

RLP is described in the Ethereum Yellow Paper, formally.

definition of trees

We define the set of possible structures \mathbb{T} :

$$(176) \quad \mathbb{T} \equiv \mathbb{L} \uplus \mathbb{B}$$

$$(177) \quad \mathbb{L} \equiv \{\mathbf{t} : \mathbf{t} = (\mathbf{t}[0], \mathbf{t}[1], \dots) \wedge \forall n < \|\mathbf{t}\| : \mathbf{t}[n] \in \mathbb{T}\}$$

$$(178) \quad \mathbb{B} \equiv \{\mathbf{b} : \mathbf{b} = (\mathbf{b}[0], \mathbf{b}[1], \dots) \wedge \forall n < \|\mathbf{b}\| : \mathbf{b}[n] \in \mathbb{O}\}$$

Where \mathbb{O} is the set of (8-bit) bytes. Thus \mathbb{B} is the set of all sequences of bytes (otherwise known as byte arrays, and a leaf if imagined as a tree), \mathbb{L} is the set of all tree-like (sub-)structures that are not a single leaf (a branch node if imagined as a tree) and \mathbb{T} is the set of all byte arrays and such structural sequences. The disjoint union \uplus is needed only to distinguish the empty byte array $() \in \mathbb{B}$ from the empty list $() \in \mathbb{L}$, which are encoded differently as defined below; as common, we will abuse notation and leave the disjoint union indices implicit, inferable from context.

We define the RLP function as RLP through two sub-functions, the first handling the instance when the value is a byte array, the second when it is a sequence of further values:

$$(179) \quad \text{RLP}(\mathbf{x}) \equiv \begin{cases} R_b(\mathbf{x}) & \text{if } \mathbf{x} \in \mathbb{B} \\ R_l(\mathbf{x}) & \text{otherwise} \end{cases}$$

If the value to be serialised is a byte array, the RLP serialisation takes one of three forms:

- If the byte array contains solely a single byte and that single byte is less than 128, then the input is exactly equal to the output.
- If the byte array contains fewer than 56 bytes, then the output is equal to the input prefixed by the byte equal to the length of the byte array plus 128.
- Otherwise, the output is equal to the input, provided that it contains fewer than 2^{64} bytes, prefixed by the minimal-length byte array which when interpreted as a big-endian integer is equal to the length of the input byte array, which is itself prefixed by the number of bytes required to faithfully encode this length value plus 183.

Byte arrays containing 2^{64} or more bytes cannot be encoded. This restriction ensures that the first byte of the encoding of a byte array is always below 192, and thus it can be readily distinguished from the encodings of sequences in \mathbb{L} .

Formally, we define R_b :

$$(180) \quad R_b(\mathbf{x}) \equiv \begin{cases} \mathbf{x} & \text{if } \|\mathbf{x}\| = 1 \wedge \mathbf{x}[0] < 128 \\ (128 + \|\mathbf{x}\|) \cdot \mathbf{x} & \text{else if } \|\mathbf{x}\| < 56 \\ (183 + \|\text{BE}(\|\mathbf{x}\|)\|) \cdot \text{BE}(\|\mathbf{x}\|) \cdot \mathbf{x} & \text{else if } \|\mathbf{x}\| < 2^{64} \\ \emptyset & \text{otherwise} \end{cases}$$

$$(181) \quad \text{BE}(x) \equiv (b_0, b_1, \dots) : b_0 \neq 0 \wedge x = \sum_{n=0}^{\|\mathbf{b}\|-1} b_n \cdot 256^{\|\mathbf{b}\|-1-n}$$

$$(182) \quad (x_1, \dots, x_n) \cdot (y_1, \dots, y_m) = (x_1, \dots, x_n, y_1, \dots, y_m)$$

encoding of all trees

encoding of leaf trees

RLP is described in the Ethereum Yellow Paper, formally.

there is no explicit
definition of decoding:
it goes without saying
that decoding is
the inverse of encoding

encoding of branching trees

the length of the byte array plus 128.

- Otherwise, the output is equal to the input, provided that it contains fewer than 2^{64} bytes, prefixed by the minimal-length byte array which when interpreted as a big-endian integer is equal to the length of the input byte array, which is itself prefixed by the number of bytes required to faithfully encode this length value plus 183.

Byte arrays containing 2^{64} or more bytes cannot be encoded. This restriction ensures that the first byte of the encoding of a byte array is always below 192, and thus it can be readily distinguished from the encodings of sequences in \mathbb{L} .

Formally, we define R_b :

$$(180) \quad R_b(\mathbf{x}) \equiv \begin{cases} \mathbf{x} & \text{if } \|\mathbf{x}\| = 1 \wedge \mathbf{x}[0] < 128 \\ (128 + \|\mathbf{x}\|) \cdot \mathbf{x} & \text{else if } \|\mathbf{x}\| < 56 \\ (183 + \|\text{BE}(\|\mathbf{x}\|)\|) \cdot \text{BE}(\|\mathbf{x}\|) \cdot \mathbf{x} & \text{else if } \|\mathbf{x}\| < 2^{64} \\ \emptyset & \text{otherwise} \end{cases}$$

$$(181) \quad \text{BE}(x) \equiv (b_0, b_1, \dots) : b_0 \neq 0 \wedge x = \sum_{n=0}^{\|\mathbf{b}\|-1} b_n \cdot 256^{\|\mathbf{b}\|-1-n}$$

$$(182) \quad (x_1, \dots, x_n) \cdot (y_1, \dots, y_m) = (x_1, \dots, x_n, y_1, \dots, y_m)$$

Thus BE is the function that expands a non-negative integer value to a big-endian byte array of minimal length and the dot operator performs sequence concatenation.

If instead, the value to be serialised is a sequence of other items then the RLP serialisation takes one of two forms:

- If the concatenated serialisations of each contained item is less than 56 bytes in length, then the output is equal to that concatenation prefixed by the byte equal to the length of this byte array plus 192.
- Otherwise, the output is equal to the concatenated serialisations, provided that they contain fewer than 2^{64} bytes, prefixed by the minimal-length byte array which when interpreted as a big-endian integer is equal to the length of the concatenated serialisations byte array, which is itself prefixed by the number of bytes required to faithfully encode this length value plus 247.

Sequences whose concatenated serialized items contain 2^{64} or more bytes cannot be encoded. This restriction ensures that the first byte of the encoding does not exceed 255 (otherwise it would not be a byte).

Thus we finish by formally defining R_l :

$$(183) \quad R_l(\mathbf{x}) \equiv \begin{cases} (192 + \|\mathbf{s}(\mathbf{x})\|) \cdot \mathbf{s}(\mathbf{x}) & \text{if } \mathbf{s}(\mathbf{x}) \neq \emptyset \wedge \|\mathbf{s}(\mathbf{x})\| < 56 \\ (247 + \|\text{BE}(\|\mathbf{s}(\mathbf{x})\|)\|) \cdot \text{BE}(\|\mathbf{s}(\mathbf{x})\|) \cdot \mathbf{s}(\mathbf{x}) & \text{else if } \mathbf{s}(\mathbf{x}) \neq \emptyset \wedge \|\mathbf{s}(\mathbf{x})\| < 2^{64} \\ \emptyset & \text{otherwise} \end{cases}$$

$$(184) \quad \mathbf{s}(\mathbf{x}) \equiv \begin{cases} \text{RLP}(\mathbf{x}[0]) \cdot \text{RLP}(\mathbf{x}[1]) \cdot \dots & \text{if } \forall i : \text{RLP}(\mathbf{x}[i]) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

RLP trees, in ACL2.

```
(fty::deftypes rlp-trees
  (fty::deftagsum rlp-tree
    (:leaf ((bytes byte-list)))
    (:branch ((subtrees rlp-tree-list))))
  (fty::deflist rlp-tree-list :elt-type rlp-tree))
```

$$\mathbb{T} \equiv \mathbb{L} \uplus \mathbb{B}$$

$$\mathbb{L} \equiv \{\mathbf{t} : \mathbf{t} = (\mathbf{t}[0], \mathbf{t}[1], \dots) \wedge \forall n < \|\mathbf{t}\| : \mathbf{t}[n] \in \mathbb{T}\}$$

$$\mathbb{B} \equiv \{\mathbf{b} : \mathbf{b} = (\mathbf{b}[0], \mathbf{b}[1], \dots) \wedge \forall n < \|\mathbf{b}\| : \mathbf{b}[n] \in \mathbb{O}\}$$

RLP encoding, in ACL2.

```
(define rlp-encode-bytes ((bytes byte-listp))
  :returns (mv (error? booleanp) (encoding byte-listp))
  (b* ((bytes (byte-list-fix bytes)))
    (cond ((and (= (len bytes) 1) (< (car bytes) 128)) (mv nil bytes))
          ((< (len bytes) 56) (mv nil (cons (+ 128 (len bytes)) bytes)))
          ((< (len bytes) (expt 2 64))
           (b* ((be (nat=>bebytes* (len bytes))))
             (mv nil (cons (+ 183 (len be)) (append be bytes))))))
    (t (mv t nil)))))
```

$$R_b(\mathbf{x}) \equiv \begin{cases} \mathbf{x} & \text{if } \|\mathbf{x}\| = 1 \wedge \mathbf{x}[0] < 128 \\ (128 + \|\mathbf{x}\|) \cdot \mathbf{x} & \text{else if } \|\mathbf{x}\| < 56 \\ (183 + \|\text{BE}(\|\mathbf{x}\|)\|) \cdot \text{BE}(\|\mathbf{x}\|) \cdot \mathbf{x} & \text{else if } \|\mathbf{x}\| < 2^{64} \\ \emptyset & \text{otherwise} \end{cases}$$

RLP encoding, in ACL2.

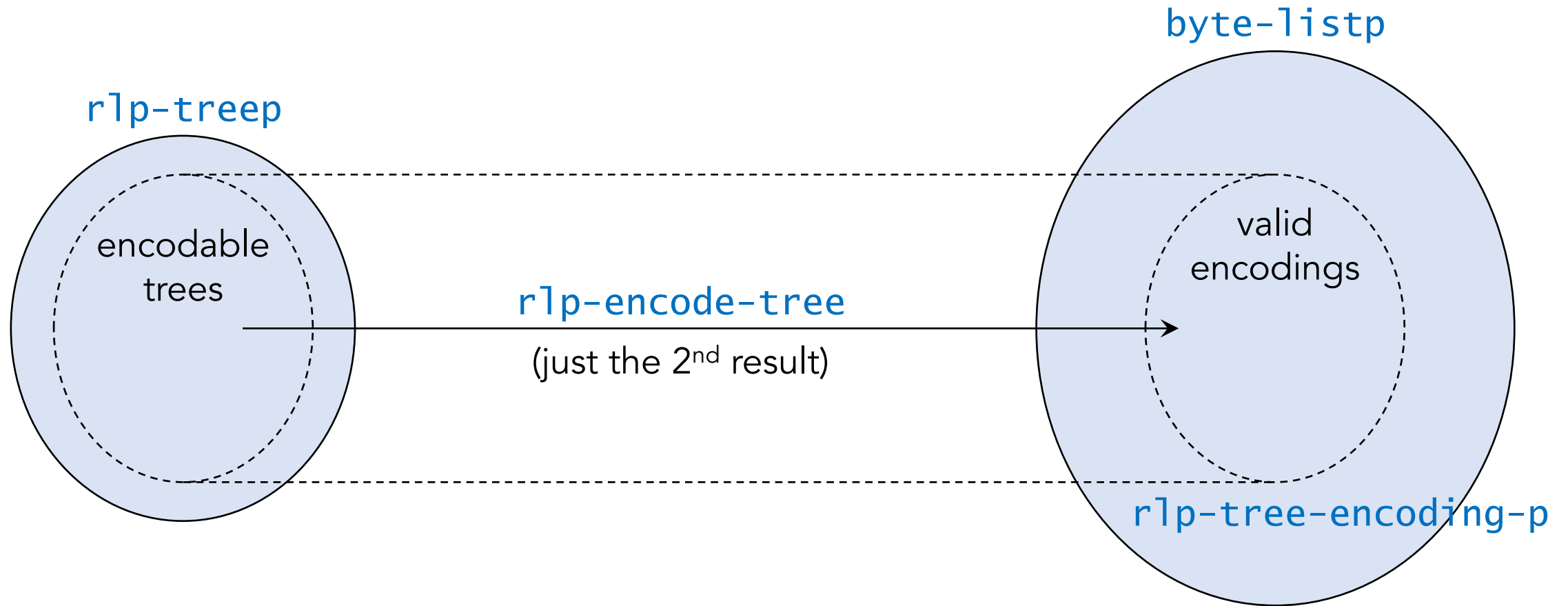
```
(define rlp-encode-tree ((tree rlp-treep))
  :returns (mv (error? booleanp) (encoding byte-listp))
  (rlp-tree-case tree
    :leaf (rlp-encode-bytes tree.bytes)
    :branch (b* (((mv error? encoding) (rlp-encode-tree-list tree.subtrees))
                  ((when error?) (mv t nil))))
              (cond ((< (len encoding) 56)
                     (mv nil (cons (+ 192 (len encoding)) encoding)))
                    ...))
  ...)

(define rlp-encode-tree-list ((trees rlp-tree-listp))
  :returns (mv (error? booleanp) (encoding byte-listp))
  (b* (((when (endp trees)) (mv nil nil))
        ...))
  ...)
```

$$\text{RLP}(\mathbf{x}) \equiv \begin{cases} R_b(\mathbf{x}) & \text{if } \mathbf{x} \in \mathbb{B} \\ R_l(\mathbf{x}) & \text{otherwise} \end{cases}$$

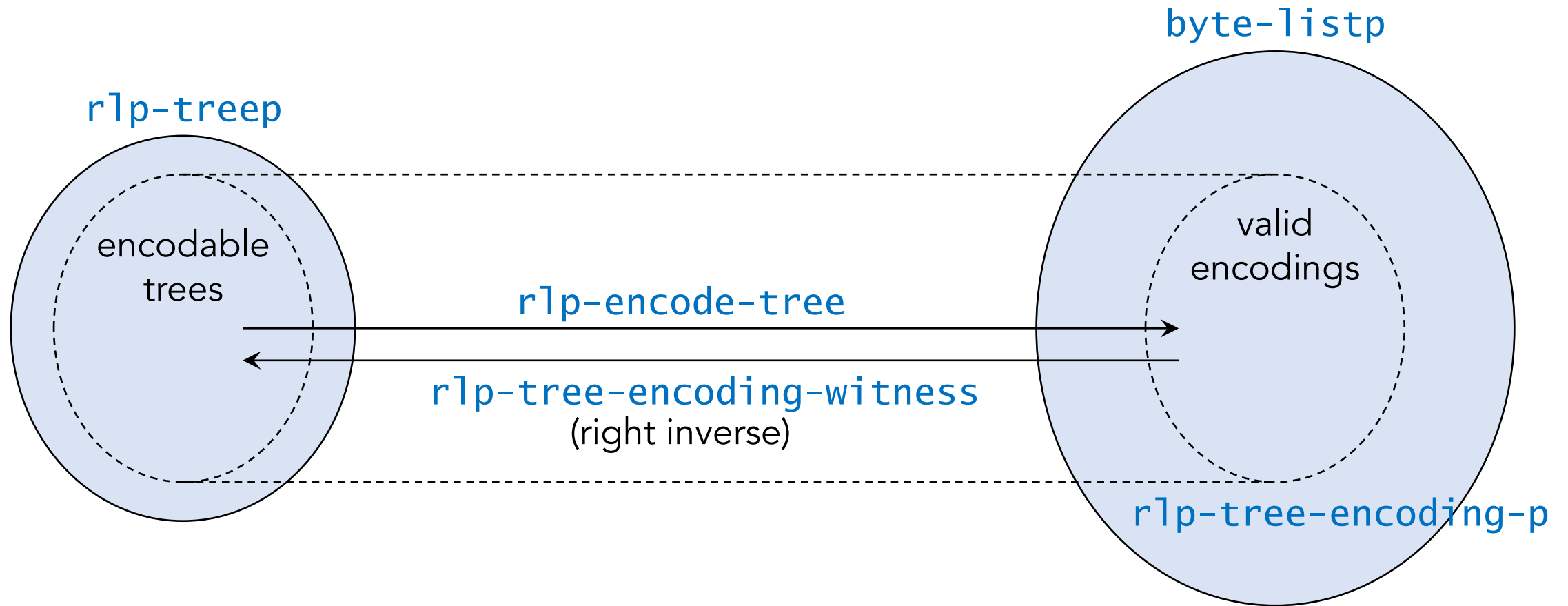
$$R_l(\mathbf{x}) \equiv \begin{cases} (192 + \|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{if } s(\mathbf{x}) \neq \emptyset \wedge \|s(\mathbf{x})\| < 56 \\ (247 + \|\text{BE}(\|s(\mathbf{x})\|)\|) \cdot \text{BE}(\|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{else if } s(\mathbf{x}) \neq \emptyset \wedge \|s(\mathbf{x})\| < 2^{64} \\ \emptyset & \text{otherwise} \end{cases}$$
$$s(\mathbf{x}) \equiv \begin{cases} \text{RLP}(\mathbf{x}[0]) \cdot \text{RLP}(\mathbf{x}[1]) \cdot \dots & \text{if } \forall i : \text{RLP}(\mathbf{x}[i]) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

RLP encoding, in ACL2.



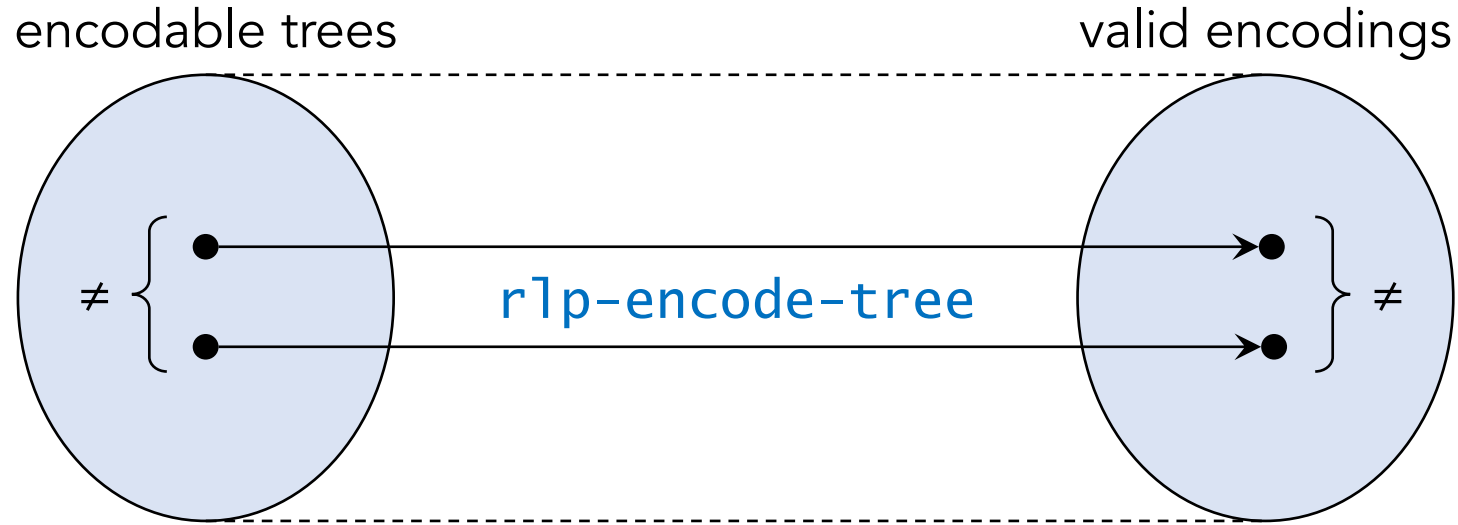
```
(define-sk rlp-tree-encoding-p ((encoding byte-list-p))
  (exists (tree) (and (rlp-tree-p tree)
    (equal (rlp-encode-tree tree)
      (mv nil (byte-list-fix encoding))))))
:skolem-name rlp-tree-encoding-witness)
```


RLP encoding, in ACL2.



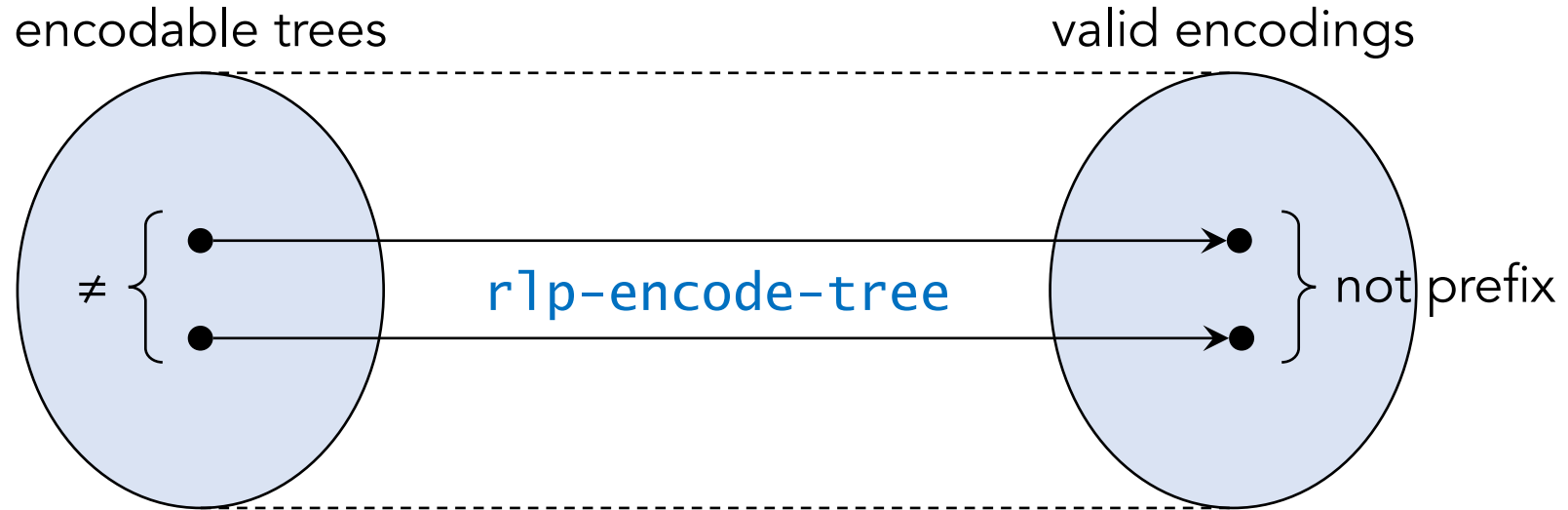
```
(define-sk rlp-tree-encoding-p ((encoding byte-listp))
  (exists (tree) (and (rlp-treep tree)
    (equal (rlp-encode-tree tree)
      (mv nil (byte-list-fix encoding))))))
:skolem-name rlp-tree-encoding-witness)
```

RLP decodability, in ACL2.



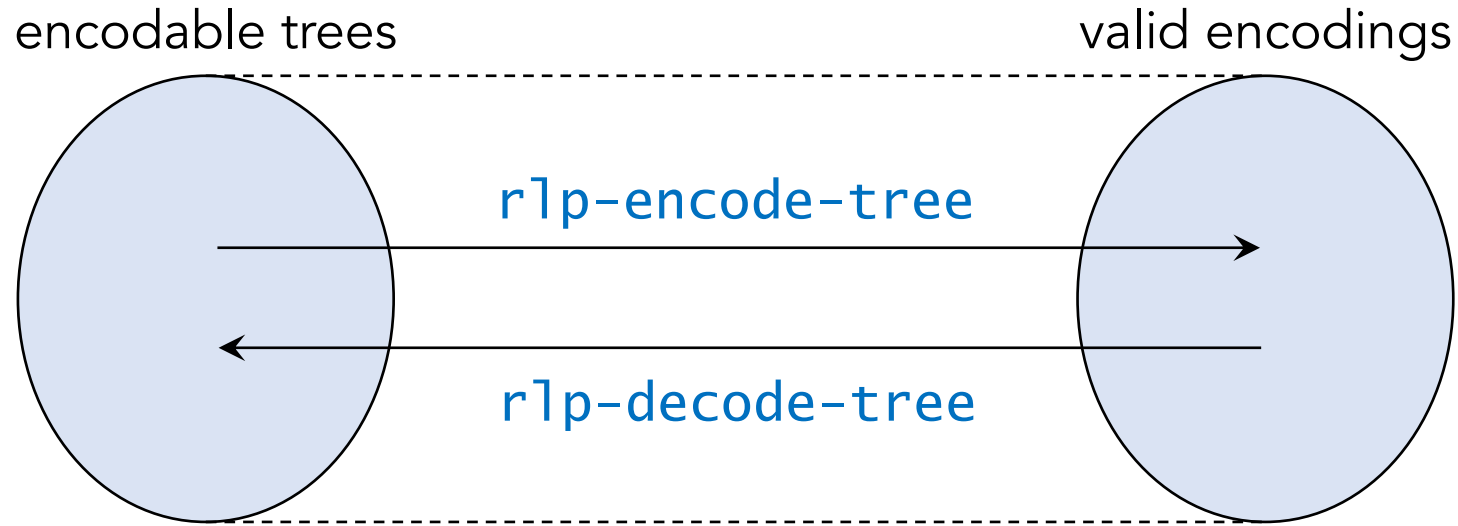
```
(defthm rlp-encode-tree-injective
  (implies (and (not (mv-nth 0 (rlp-encode-tree x)))
                 (not (mv-nth 0 (rlp-encode-tree y))))
    (equal (equal (mv-nth 1 (rlp-encode-tree x))
                  (mv-nth 1 (rlp-encode-tree y)))
      (equal (rlp-tree-fix x) (rlp-tree-fix y))))))
```

RLP decodability, in ACL2.



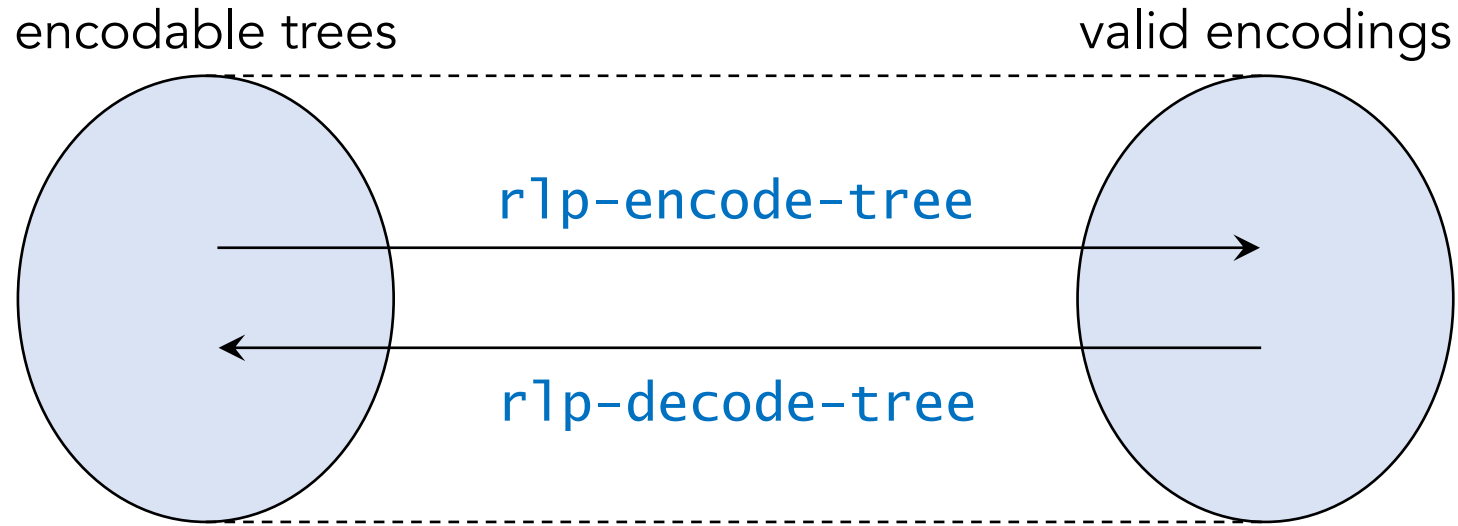
```
(defthm rlp-encode-tree-unamb-prefix
  (implies (and (not (mv-nth 0 (rlp-encode-tree x)))
                 (not (mv-nth 0 (rlp-encode-tree y))))
    (equal (prefixp (mv-nth 1 (rlp-encode-tree x))
                    (mv-nth 1 (rlp-encode-tree y)))
      (equal (mv-nth 1 (rlp-encode-tree x))
              (mv-nth 1 (rlp-encode-tree y))))))
```

RLP decoding, in ACL2, declarative.



```
(define rlp-decode-tree ((encoding byte-listp))
  :returns (mv (error? booleanp) (tree rlp-treep))
  (b* ((encoding (byte-list-fix encoding)))
    (if (rlp-tree-encoding-p encoding)
        (mv nil (rlp-tree-encoding-witness encoding))
        (mv t (rlp-tree-leaf nil)))) ; 2nd result irrelevant
```

RLP decoding, in ACL2, declarative.



```
(defthm rlp-encode-tree-of-rlp-decode-tree ; right inverse  
  ...) ; proof is straightforward, from witness axiom
```

```
(defthm rlp-decode-tree-of-rlp-encode-tree ; left inverse  
  ...) ; proof is from right inverse above and injectivity
```

RLP decoding, in ACL2, executable.

```
(define rlp-parse-tree ((encoding byte-listp))
  :returns (mv (error? maybe-rlp-error-p) (tree rlp-tree-p) (rest byte-listp))
  (b* ((encoding (byte-list-fix encoding))
      ((when (endp encoding)) ...) ; error
      ((cons first encoding) encoding)
      ((when (< first 128)) (mv nil (rlp-tree-leaf (list first)) encoding))
      ((when (<= first 183))
       (b* ((len (- first 128))
           ((when (< (len encoding) len)) ...) ; error
           (bytes (take len encoding))
           ((when (and (= len 1) (< (car bytes) 128))) ...) ; error
           (mv nil (rlp-tree-leaf bytes) (nthcdr len encoding))))
       ((when (< first 192))
        (b* ((lenlen (- first 183))
            ((when (< (len encoding) lenlen)) ...) ; error
            (len-bytes (take lenlen encoding))
            ((unless (equal (trim-bendian* len-bytes) len-bytes)) ...) ; error
            (encoding (nthcdr lenlen encoding))
            (len (bebytes=>nat len-bytes))
            ((when (<= len 55)) ...) ; error
```

RLP decoding, in ACL2, executable.

```
(len (bebytes=>nat len-bytes))
(when (<= len 55)) ... ; error
(when (< (len encoding) len)) ... ; error
(subencoding (take len encoding))
(encoding (nthcdr len encoding))
(mv error? subtrees) (rlp-parse-tree-list subencoding))
(when error?) ... ; error
(mv nil (rlp-tree-branch subtrees) encoding)))
```

```
(define rlp-parse-tree-list ((encoding byte-listp))
:returns (mv (error? maybe-rlp-error-p) (trees rlp-tree-listp))
(b* (((when (endp encoding)) (mv nil nil))
      ((mv error? tree encoding1) (rlp-parse-tree encoding))
      ((when error?) ... ; error
        (unless (mbt (< (len encoding1) (len encoding)))) ... ; error
        ((mv error? trees) (rlp-parse-tree-list encoding1))
        ((when error?) ... ; error
         (mv nil (cons tree trees))))))
```

RLP decoding, in ACL2, executable.

```
(define rlp-parse-tree ((encoding byte-listp))
  :returns (mv (error? maybe-rlp-error-p) (tree rlp-treep) (rest byte-listp))
  ...)
```

```
(define rlp-decodex-tree ((encoding byte-listp))
  :returns (mv (error? maybe-rlp-error-p) (tree rlp-treep))
  (b* (((mv error? tree rest) (rlp-parse-tree encoding))
    ((when error?) ...) ; error
    ((when (consp rest)) ...)) ; error
    (mv nil tree)))
```

; parser is (left and right) inverse of encoder:

```
(defthm rlp-parse-tree-of-rlp-encode-tree ...) ; accepts all valid encodings
(defthm rlp-encode-tree-of-rlp-parse-tree ...) ; accepts only valid encodings
```

; executable decoder is (left and right) inverse of encoder:

```
(defthm rlp-decodex-tree-of-rlp-encode-tree ...)
(defthm rlp-encode-tree-of-rlp-decodex-tree ...)
```





RLP decoding, in ACL2, executable and verified.

```
(define rlp-decodex-tree ((encoding byte-listp))
  :returns (mv (error? maybe-rlp-error-p) (tree rlp-treep))
  ...)
```

```
; executable decoder is (left and right) inverse of encoder:
(defthm rlp-decodex-tree-of-rlp-encode-tree ...)
(defthm rlp-encode-tree-of-rlp-decodex-tree ...)
```



```
; executable decoder is equivalent to declarative decoder:
(defthm rlp-decode-tree-is-rlp-decodex-tree
  (and (iff (mv-nth 0 (rlp-decode-tree encoding))
            (mv-nth 0 (rlp-decodex-tree encoding)))
    (equal (mv-nth 1 (rlp-decode-tree encoding))
            (mv-nth 1 (rlp-decodex-tree encoding))))))
```

See the RLP manual pages for much more information.

 ABCDEFGHIJKLM
NOPQRSTUVWXYZ

Jump to

Search



⇒ Top

+ ACL2

+ Books

+ Boolean-reasoning

+ Debugging

+ Documentation

+ Hardware-verification

+ Interfacing-tools

+ Macro-libraries

+ Math

+ Projects

+ Proof-automation

+ Software-verification

+ Std

+ Testing-utilities

Ethereum

Rlp

[books]/kestrel/ethereum/rlp/top.lisp

Recursive Length Prefix (RLP).

RLP is a serialization (encoding) method for Ethereum, described in [YP:B] and in [Page 'RLP' of \[Wiki\]](#); we reference that page of [Wiki] as '[Wiki:RLP]'.

Subtopics

Rlp-big-endian-representations

Big-endian representation of scalars in RLP.

Rlp-tree

RLP trees.

Rlp-encoding

RLP encoding.

Rlp-decodability

Proofs that RLP encodings can be decoded.

Rlp-decoding-declarative

Declarative definitions of RLP decoding.

Rlp-decoding-executable

Executable definitions of RLP decoding.

ETHEREUM
Package