Integer
Signed 32-bit integer.

Long
Signed 64-bit integer.

Array
An array of any logical data type: array<type>

2.2 ~~Writing bits to a bit stream~~Reading and writing bits in a bit stream

The CORE block supports bit-based encoding methods. A bit stream consists of a sequence of 1s and 0s. The bits are written most significant bit first where new bits are stacked to the right and full bytes on the left are written out. In a bit stream the last byte will be incomplete if less than 8 bits have been written to it. In this case the bits in the last byte are shifted to the left to complete a whole byte.

Example of writing to bit stream

Let's consider the following example. The table below shows a sequence of write operations:

Operation order	Buffer state before	Written bits	Buffer state after	Issued bytes
1	~~0x0~~ xxxx xxxx	1	~~0x1~~ xxxx xxx1 (0x01)	-
2	~~0x1~~ xxxx xxxx	0	~~0x2~~ xxxx xx10 (0x02)	-
3	~~0x2~~ xxxx xx10	11	~~0xB~~ xxxx 1011 (0x0B)	-
4	~~0xB~~ xxxx 1011	0000 0111	~~0x7~~ xxxx 0111 (0x07)	1011 0000 (0xB0)

After flushing the above bit stream the following bytes are written: 0xB0 0x70. Please note that the last byte was 0x7 before shifting to the left and became 0x70 after that:

```
> echo "obase=16; ibase=2; 00000111" | bc
7

> echo "obase=16; ibase=2; 01110000" | bc
70
```

And the whole bit sequence:

```
> echo "obase=2; ibase=16; B070" | bc
1011000001110000
```

When reading the bits from the bit sequence~~it must be known that only~~ , only the first 12 bits are meaningful and the ~~bit stream should not be read after that~~remaining 4 will should be discarded.

Note on reading from and writing to bit stream

When reading and writing to a bit stream ~~both the value and the number of bits in the value must be known. This is because programming languages normally operate with bytes (8 bits) and to specify which bits are to be written requires a bit-holder, for example an integer, and~~ our numeric values are typically held in a byte oriented data type, such as an 8-bit or 32-bit integer. The bit stream itself does not explicitly store the number of bits ~~in it . Equally, when reading a value from a bit stream the number of bits must be known in advance. In case of prefix codes (e.g. Huffman) all possible bit combinations are either known in advance or it is possible to calculate how many bits will follow based on the first few bits. Alternatively, two codes can be combined, where the first contains the number of bits to read.~~ per value, and it will vary by context, so we must know this by other means. For example, we may be reading bits using a BETA encoding whose parameters indicate each value is 6 bits. So we read the next 6 bits into a 32-bit integer to get a value between 0 and 63. The next bits may be for a HUFFMAN encoding, in which case we can read one bit at a time until we match a known code-word in the Huffman tree.

2.3 Writing bytes to a byte stream

Byte streams cannot be mixed in the same block as bit streams. The interpretation of byte stream is straightforward. CRAM uses *little endianness* for bytes when applicable and defines the following storage data types:

Boolean (bool)
Boolean is written as 1-byte with 0x0 being 'false' and 0x1 being 'true'.

Integer (int32)
Signed 32-bit integer, written as 4 bytes in little-endian byte order.

Long (int64)
Signed 64-bit integer, written as 8 bytes in little-endian byte order.

ITF-8 integer (itf8)
This is an alternative way to write an integer value. The idea is similar to UTF-8 encoding and therefore this encoding is called ITF-8 (Integer Transformation Format - 8 bit).

The most significant bits of the first byte have special meaning and are called 'prefix'. These are 0 to 4 true bits followed by a 0. The number of 1's denote the number of bytes to follow. To accommodate 32 bits such representation requires 5 bytes with only 4 lower bits used in the last byte 5.

LTF-8 long (ltf8)
See ITF-8 for more details. The only difference between ITF-8 and LTF-8 is the number of bytes used to encode a single value. To do so 64 bits are required and this can be done with 9 byte at most with the first byte consisting of just 1s or 0xFF value.

Array (array<type>)
A variable sized array with an explicitly written dimension. Array length is written first as integer (itf8), followed by the elements of the array.

Implicit or fixed-size arrays are also used, written as *type*`[ ]` or *type*`[4]` (for example). These have no explicit dimension included in the file format and instead rely on the specification itself to document the array size.

Encoding
Encoding is a data type that specifies how data series have been compressed. Encodings are defined as encoding<type> where the type is a logical data type as opposed to a storage data type.

An encoding is written as follows. The first integer (itf8) denotes the codec id and the second integer (itf8) the number of bytes in the following encoding-specific values.

Subexponential encoding example:

Value	**Type**	**Name**
0x7	itf8	codec id
0x2	itf8	number of bytes to follow
0x0	itf8	offset
0x1	itf8	K parameter

The first byte "0x7" is the codec id.

The next byte "0x2" denotes the length of the bytes to follow (2).

The subexponential encoding has 2 parameters: integer (itf8) offset and integer (itf8) K.

offset = 0x0 = 0

K = 0x1 = 1

Map
A map is a collection of keys and associated values. A map with N keys is written as follows:

size in bytes	N	key 1	value 1	key ...	value ...	key N	value N

Both the size in bytes and the number of keys are written as integer (itf8). Keys and values are written according to their data types and are specific to each map.

String
A string is represented as byte arrays using UTF-8 format. Read names, reference sequence names and tag values with type 'Z' are stored as UTF-8.