Protobuf, or protocol buffers, is a message description language and serialization protocol developed by Google. It is similar to other descriptive data formats like ASN1, and is commonly used as part of the gRPC remote procedure call system. Protobuf and gRPC are one of the most popular serialization formats.

0.1 The `proto` Language

A high-level protobuf message description is written in a `.proto` file in the `proto` language. Each message is described by a *message descriptor* which declares each field and its corresponding type.

```
syntax = "proto3";

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 results_per_page = 3;
}
```

Figure 1: Example protobuf message description from the protobuf documentation [**LanguageGuideProto**].

In figure 1, a message called `SearchRequest` is declared. This message contains three fields, the query, page number to display and number of results per page. Each field is assigned a *field number*, which is used in the serialized version of the message. Each field number must be unique, as they are used to identify the corresponding binary blob in the encoded message. A `.proto` file can define multiple messages, and messages can be embedded in another by using the name of that message as the type of a field in the other message.

Each field is defined with a type, with the primitive types summarized in table 1.

Type	Notes
`double`	Standard 64 bit double-precision floating point number.
`float`	Standard 32 bit floating point number.
`int32`	Variable-length signed 32 bit integer.
`int64`	Variable-length signed 64 bit integer.
`uint32`	Variable-length unsigned 32 bit integer.
`uint64`	Variable-length unsigned 64 bit integer.
`sint32`	Variable-length 32 bit integer with more efficient encoding for negative numbers.
`sint64`	Variable-length 64 bit integer with more efficient encoding for negative numbers.
`fixed32`	Four byte integer, more efficient than `uint32` for values greater than 2^{38}.
`fixed64`	Eight byte integer, more efficient than `uint64` for values greater than 2^{56}.
`sfixed32`	Four byte signed integer.
`sfixed64`	Eight byte signed integer.
`bool`	Boolean, encoded in one byte.
`string`	A string of UFT-8 or 7-bit ASCII characters shorter than 2^{32} bytes.
`bytes`	An arbitrary sequence of less than 2^{32} bytes.

Table 1: Summary of the protobuf scalar values.

Any protobuf field can marked as `optional` or `repeated`. An `optional` field can be either set to a value or omitted from the message. Similarly, a `repeated` field can be repeated in the message zero or more times, functioning as a list now. A field without a cardinality modifier is known as an implicit field and can also be omitted from a message, just like an `optional` field. During de-serialization, a type-specific default value will be used (this is consistent with accessing the value of an unset `optional` field). The only difference between an implicit field and `optional` field is that the optional field can differentiate between an unset field and field set to the default value by providing a method which reports if the `optional` field is set.

Since the protobuf language is designed to be both forward and backwards compatible, it is possible to remove or add fields. When removing fields, it is encouraged to reserve the field number to prevent reuse of the field in a future. It is also possible to reserved field names, which isn't important for the protobuf binary encoding format, this is important if you're using the JSON encoding of a protobuf message.

```
syntax = "proto3";

message Foo {
  reserved 2, 15, 9 to 11;
  reserved "foo", "bar";
}
```

Figure 2: Example protobuf message using reserved fields [**LanguageGuideProto**].

Protobuf also supports enumerations, for when a field may only take a finite number of values. Due to how default values work in protobuf, enums are required to have a value defined with a value zero and it should have either "UNSPECIFIED" or "UNKNOWN" since that will be the default value.

```
syntax = "proto3";

enum {
  CORPUS_UNSPECIFIED = 0;
  CORPUS_UNIVERSAL = 1;
  CORPUS_WEB = 2;
  CORPUS_IMAGES = 3;
  CORPUS_LOCAL = 4;
  CORPUS_NEWS = 5;
  CORPUS_PRODUCTS = 6;
  CORPUS_VIDEO = 7;
}

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 results_per_page = 3;
  Corpus corpus = 4;
}
```

Figure 3: Example protobuf message using an enum [**LanguageGuideProto**].

There are two other features which need to be modeled, `map` and `oneof`. Maps can be thought of like a standard map in `go` or a `dict` in `python`. From a serialization perspective, this is an unordered map that could be modeled as a list of key-value pairs. A `map` field cannot be `repeated`, and repeated keys have the last occurrence win, as is standard with protobuf (see Section 0.2).

```
message Foo {
  map<int32, string> map_field = 1;
}
```

```
message MapFieldEntry {
  int32 key = 1;
  string value = 2;
}

message Foo {
  repeated MapFieldEntry map_field = 1;
}
```

Figure 4: Example of `map` syntax and the corresponding wire-equivalent syntax [**LanguageGuideProto**].

The final important feature of the proto language is a `oneof` field, which operates like a union in `C`. Setting any member of the `oneof` will clear any previously set field. A `oneof` field cannot contain a `repeated` field or a `map` and the `oneof` field itself cannot be `repeated`.

```
syntax = "proto3";

message User {
  oneof user_id {
    string email = 4;
    int32 phone = 2;
  }
}
```

Figure 5: Example protobuf message using an enum [**LanguageGuideProto**].

0.2 The Protobuf Encoding

Complimenting the Protobuf schema language is the protobuf encoding format. All valid protobuf messages written the the proto language will be eventually encoding into a binary blob in respecting this encoding.

0.2.1 Variable-Length Integers

The heart of the protobuf encoding is the variable-length integer encoding, which represents a 32 or 64 bit integer in between one and ten bytes. The encoding format itself is relatively simple. The bits of the integer to be encoded are grouped into bundles of seven. The bundles are ordered in little endian order with the first bit of each byte set to one if the next byte is also part of the same integer.

For example, consider the number 163, or 10100011 in binary. Splitting these into bundles of seven bits produces "0000001" and "0100011". These are reordered into little endian order, "0000001" then "0100011". Finally set the continuation bit on the first byte of the encoding and concatenate them together to get "10000010100011".

0.2.2 Field Identifiers

A message is serialized as a sequence of fields, in any order. Each field is encoded as a *header-length-value* structure, although not every field has a dedicated length in the encoding. The tag consists of the field number as defined in the `proto` file, encoded in the variable length integer encoding discussed above. However, the tag isn't just the field number, it also includes information on the type of payload as detailed in Table 2. The tag is stored in the last three bits of the field number varint, effectively encoded as `(field_number « 3) | tag`.

ID	Name	Uses
0	`VARINT`	`int32`, `int64`, `uint32`, `uint64`, `sint32`, `sint64`, `bool`, `enum`
1	`I64`	`fixed64`, `sfixed64`, `double`
2	`LEN`	`string`, `bytes`, embedded messages, packed repeated fields
5	`I32`	`fixed32`, `sfixed32`, `float`

Table 2: Description of `proto` tag values. Note that values 3 and 4 correspond to `SGROUP` and `EGROUP`, tags used in protobuf version 2 and deprecated in protobuf 3 [**Encoding**].

0.2.3 Length Delimited Fields

Nested messages, string and bytes are encoded with their payload length in bytes as a varint just after the tag. Despite being variable-length, varint fields aren't encoded with an explicit length since checking the length of the varint doesn't provide significance benefits when the maximum length varint is only 10 bytes. Lengths are encoded in bytes so that deserializers can skip large unknown fields efficiently and know when exactly the unknown field ends.

All variable length fields except varints are prefixed with a length and then the rest of the payload, with nested messages simply beginning with the tag for their first field and continuing from there. Since nested messages include the complete field encodings, it is important for the parser to be able to skip an unknown field exactly rather than incorrectly thinking a nested unknown field is setting field in the parent message.

0.2.4 Signed Integers

The standard varint encoding is only for unsigned integers. Signed integers can be encoded in both `int32` and `int64` or `sint32` and `sint64` field types. The `intN` types uses standard two's complement to encode negative integers. However, since a negative two's compliment integer always has the sign bit set, any negative number will be encoded in a maximum length varint 10 bytes long.

On the other hand, the `sintN` types use a "ZigZag" encoding to more efficiently represent smaller negative numbers [**Encoding**]. In this encoding, a positive value p is encoded as $2 \times p$ while a negative value n is encoded as $2 \times |n| - 1$. Using bit-shift operators, this would be `(n << 1) ^ (n >> 31)` or the naturally extended 64-bit version.

Signed Original	Zig-Zag Encoding
0	0
-1	1
1	2
-2	3
⋮	⋮

Table 3: Some zig zag encoded integers

While the zig-zag encoding does save space for smaller negative values in particular, it does increase the amount of space required to encode large positive values, which is why it isn't the standard encoding for signed integers.