Open Data Description Language Specification

Version 3.0

Updated January 4, 2021

by Eric Lengyel

Terathon Software LLC
Lincoln, California

1 Introduction

The Open Data Description Language (OpenDDL) is a generic text-based language that is designed to store arbitrary data in a concise human-readable format. It can be used as a means for easily exchanging information among many programs or simply as a method for storing a program's data in an editable format. Each unit of data in an OpenDDL file has an explicitly specified type, and this eliminates ambiguity and fragile inferencing methods that can impact the integrity of the data. This strong typing is further supported by the specification of an exact number of bits required to store numerical data values when converted to a binary representation.

The data structures in an OpenDDL file are organized as a collection of trees. The language includes a built-in mechanism for making references from one data structure to any other data structure, effectively allowing the contents of a file to take the form of a directed graph.

As a foundation for higher-level data formats, OpenDDL is intended to be minimalistic. It assigns no meaning whatsoever to any data beyond its hierarchical organization, and it imposes no restrictions on the composition of data structures. Semantics and validation are left to be defined by specific higher-level formats derived from OpenDDL. The core language is designed to place as little burden as possible on readers so that it's easy to write programs that understand OpenDDL.

The OpenDDL syntax is illustrated in the "railroad diagrams" found throughout this specification, and it is designed to feel familiar to C/C++ programmers. Whitespace never has any meaning, so OpenDDL files can be formatted in any manner preferred.

2 Structures

An OpenDDL file is composed of a sequence of *structures*. A single structure consists of a type identifier followed by an optional name, an optional list of properties, and its data payload enclosed in braces, as shown in Figure 1. There are two general classes of structures called *primitive* structures and *derived* structures. Primitive structures have types that are defined by OpenDDL itself, and they contain primitive data such as integers, floating-point numbers, or strings. Derived structures represent custom data types defined by a derivative file format, and they can contain other structures, primitive or derived, that can be organized in a hierarchical manner.

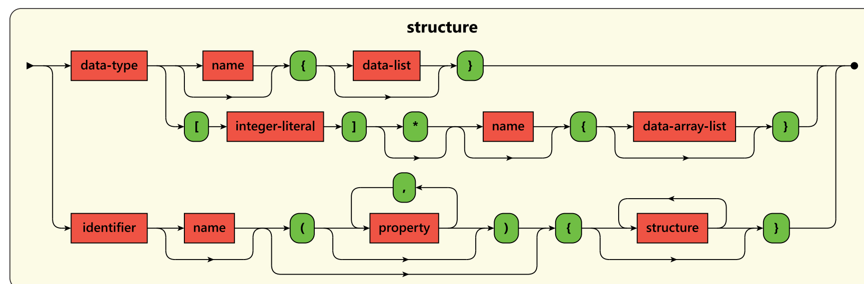

Figure 1. An OpenDDL file contains a sequence of structures that follow the production rule shown here.

[*Example* — Suppose that a derivative file format defined a data type called `Vertex` that contains the 3D coordinates of a single vertex position. This could be written as follows.

```
Vertex
{
    float {1.0, 2.0, 3.0}
}
```

The `Vertex` identifier represents a derived structure defined by the file format, and it contains another structure of type `float`, which is a primitive data type defined by OpenDDL. The data in the `float` structure consists of the three values 1.0, 2.0, and 3.0. — *End example*]

If a structure has a type that is not recognized by an implementation, then that structure and all of the data it contains must be ignored without producing an error. This allows extensions to be added to a data format without breaking compatibility with implementations that do not support them.

3 Data Types

OpenDDL defines the 16 *primitive data types* listed in Table 1, and they can be specified by the long identifiers and short identifiers shown in Figure 2. There is no difference in the meaning between the long and short identifiers, so they can be used interchangeably. The three floating-point data types each have additional long and short identifiers, making it possible to specify those types with four different identifiers that all have equivalent meanings.

When used as the identifier for a structure, each entry in the Table 1 indicates that the structure is a primitive structure and its data payload is composed of an array of literal values. Primitive structures cannot have substructures.

Long Identifier	Short Identifier	Description
`bool`	`b`	Boolean type that can have the value `true` or `false`.
`int8`	`i8`	8-bit signed integer that can have values in the range $[-2^7, 2^7 - 1]$.
`int16`	`i16`	16-bit signed integer that can have values in the range $[-2^{15}, 2^{15} - 1]$.
`int32`	`i32`	32-bit signed integer that can have values in the range $[-2^{31}, 2^{31} - 1]$.
`int64`	`i64`	64-bit signed integer that can have values in the range $[-2^{63}, 2^{63} - 1]$.
`uint8`	`u8`	An 8-bit unsigned integer that can have values in the range $[0, 2^8 - 1]$.
`uint16`	`u16`	16-bit unsigned integer that can have values in the range $[0, 2^{16} - 1]$.
`uint32`	`u32`	32-bit unsigned integer that can have values in the range $[0, 2^{32} - 1]$.
`uint64`	`u64`	64-bit unsigned integer that can have values in the range $[0, 2^{64} - 1]$.
`half`, `float16`	`h`, `f16`	16-bit floating-point type in the standard S1-E5-M10 format.
`float`, `float32`	`f`, `f32`	32-bit floating-point type in the standard S1-E8-M23 format.
`double`, `float64`	`d`, `f64`	64-bit floating-point type in the standard S1-E11-M52 format.
`string`	`s`	Double-quoted character string with contents encoded in UTF-8.
`ref`	`r`	Sequence of structure names, or the keyword `null`.
`type`	`t`	Type whose values are identifiers naming the types in this table.
`base64`	`z`	Generic binary data encoded as base64.

Table 1. These are the 16 primitive data types defined by OpenDDL.

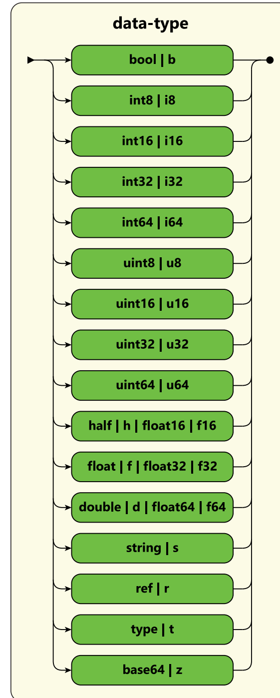

Figure 2. These are the 16 primitive data types defined by OpenDDL.

There is no implicit type conversion in OpenDDL. Data belonging to a primitive structure must be directly parsable as literal values corresponding to the structure's data type.

The `type` data type is convenient for schemas built upon OpenDDL itself in order to define valid type usages in derivative file formats.

3.1 Booleans

A boolean value is one of the keywords `false` or `true`, as shown in Figure 3. The numerical values `0` and `1` may also be specified, and they are equivalent to `false` or `true`, respectively.

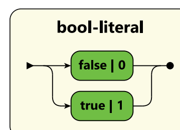

Figure 3. A boolean value is one of the keywords `false` or `true` or one of the equivalent numerical values `0` or `1`.

3.2 Integers

Integers can be specified as a decimal number, a hexadecimal number, an octal number, a binary number, or a single-quoted character literal.

Between any two consecutive digits of each type of integer literal, a single underscore character may be inserted as a separator to enhance readability. The presence of underscore characters and their positions have no significance, and they do not affect the value of a literal.

A decimal literal is simply composed of a sequence of numerical digits, as shown in Figure 4, and leading zeros are permitted.

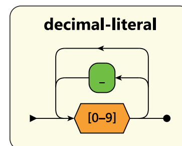

Figure 4. A decimal literal is any sequence of numerical digits.

A hexadecimal literal is specified by prefixing a number with `0x` or `0X`, as shown in Figure 5. This is followed, without any intervening whitespace, by any number of hexadecimal digits, shown in Figure 6, that don't cause the underlying integer type to overflow. The letters A–F in a hexadecimal literal are not case sensitive.

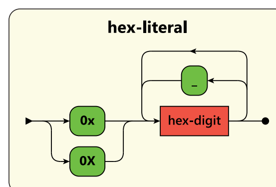

Figure 5. A hexadecimal literal starts with `0x` or `0X` and continues with one or more hexadecimal digits.

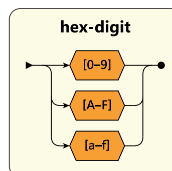

Figure 6. A hexadecimal digit is a numerical digit 0–9 or a letter A–F (with no regard for case).

An octal literal is specified by prefixing a number with `0o` or `0O`, as shown in Figure 7. This is followed, without any intervening whitespace, by any number of digits between 0 and 7, inclusive, that don't cause the underlying integer type to overflow.

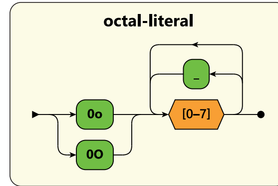

Figure 7. An octal literal starts with `0o` or `0O` and continues with one or more octal digits.

A binary literal is specified by prefixing a number with `0b` or `0B`, as shown in Figure 8. This is followed, without any intervening whitespace, by any number of zeros and ones that don't cause the underlying integer type to overflow.

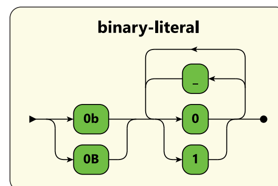

Figure 8. A binary literal starts with `0b` or `0B` and continues with one or more binary digits.

A character literal is specified by a sequence of printable ASCII characters enclosed in single quotes, as shown in Figure 9. OpenDDL supports the escape sequences listed in Table 2 and illustrated in Figure 10. Escape sequences may be used to generate control characters or arbitrary byte values. The single quote (`'`) and backslash (`\`) characters cannot be represented directly and must be encoded with escape sequences. The `\x` escape sequence is always followed by exactly two hexadecimal digits. Each character (after resolving escape sequences) corresponds to exactly one byte in the resulting integer value, and the right-most character corresponds to the least significant byte.

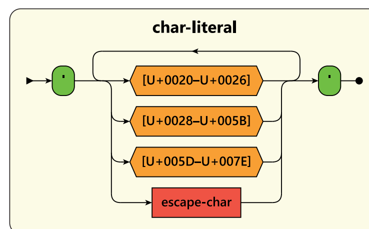

Figure 9. A character literal is composed of a sequence of printable ASCII characters enclosed in single quotes. The single quote (`'`) and backslash (`\`) characters cannot be represented directly and must be encoded with escape sequences.

Escape Sequence	ASCII Code	Description
`\"`	`0x22`	Double quote
`\'`	`0x27`	Single quote
`\?`	`0x3F`	Question mark
`\\`	`0x5C`	Backslash
`\a`	`0x07`	Bell
`\b`	`0x08`	Backspace
`\f`	`0x0C`	Formfeed
`\n`	`0x0A`	Newline
`\r`	`0x0D`	Carriage return
`\t`	`0x09`	Horizontal tab
`\v`	`0x0B`	Vertical tab
`\x`*hh*	–	Byte value specified by the two hex digits *hh*

Table 2. These are the escape sequences supported by OpenDDL for character literals.

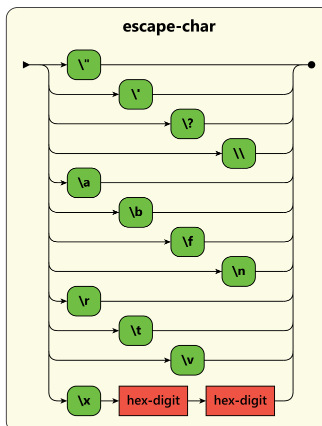

Figure 10. An escape character consists of a backslash (`\`) followed by a single character code. In the case of the `\x` character code, the escape sequence includes exactly two additional hexadecimal digits.

An integer literal is composed of an optional plus or minus sign followed by a decimal, hexadecimal, octal, binary, or character literal, as shown in Figure 11.

[*Example* — In the following code, the same 32-bit unsigned integer value is repeated five times using different literal types: a decimal literal, a hexadecimal literal, an octal literal, a binary literal, and a character literal.

```
uint32
{
    1094861636,
    0x41424344,
    0o10120441504,
    0b0100_0001_0100_0010_0100_0011_0100_0100,
    'ABCD'
}
```

— *End example*]

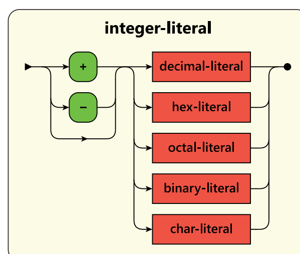

Figure 11. An integer literal is composed of an optional sign followed by a decimal, hexadecimal, octal, binary, or character literal.

3.3 Floating-Point Numbers

Floating-point numbers can be specified as a decimal number with or without a decimal point and fraction, and with or without a trailing exponent, as shown in Figure 12. Floating-point numbers may also be specified as hexadecimal, octal, or binary literals representing the underlying bit pattern of the number. This is particularly useful for lossless exchange of floating-point data since round-off errors possible in the conversion to and from a decimal representation are avoided. Using a hexadecimal, octal, or binary representation is also the only way to specify a floating-point infinity or not-a-number (NaN) value.

As with integer literals, an underscore character may be inserted between any two consecutive numerical digits in a floating-point literal to enhance readability. Underscore characters are ignored and do not affect the value of a literal.

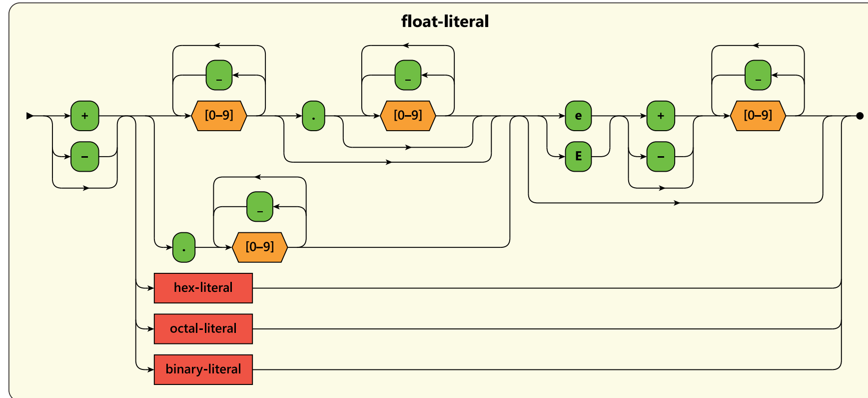

Figure 12. A floating-point literal is composed of an optional sign followed by a number with or without a decimal point and an optional exponent. Hexadecimal, octal, and binary literals representing the underlying bit pattern are also accepted.

3.4 Strings

Strings are composed of a sequence of characters enclosed in double quotes, as shown in Figure 13. Unicode values (encoded as UTF-8) in the following ranges may be directly included in a string literal:

- [`U+0020`, `U+0021`]
- [`U+0023`, `U+005B`]
- [`U+005D`, `U+007E`]
- [`U+00A0`, `U+D7FF`]
- [`U+E000`, `U+FFFD`]
- [`U+010000`, `U+10FFFF`]

This is the only place where non-ASCII characters are allowed other than in comments.

A string may contain the escape sequences defined for character literals (see Figure 10). The double quote (`"`) and backslash (`\`) characters cannot be represented directly and must be encoded with escape sequences. String literals also support the `\u` escape sequence, which specifies a nonzero Unicode character using exactly four hexadecimal digits immediately following the `u`. In order to support Unicode characters outside the Basic Multilingual Plane (BMP), a six-digit code can be specified by using an uppercase `U`. The `\U` escape sequence must be followed by exactly six hexadecimal digits that specify a value in the range [`0x000001`, `0x10FFFF`].

Multiple string literals may be placed adjacent to each other with or without intervening whitespace, and this results in concatenation.

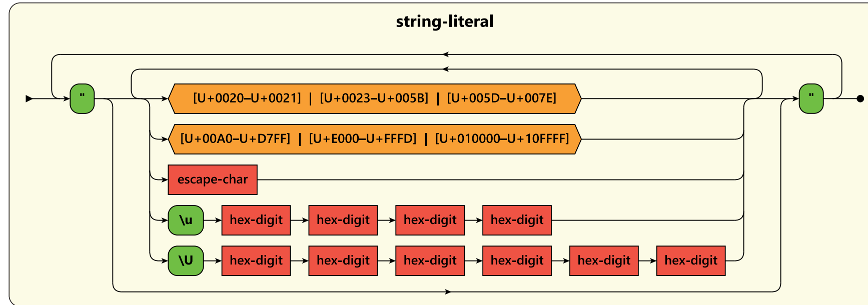

Figure 13. A string literal is composed of a sequence of Unicode characters enclosed in double quotes. The double-quote ("), backslash (\), and non-printing control characters are excluded from the set of characters that can be directly represented. A string may contain the same escape characters as a character literal as well as additional Unicode escape sequences. Adjacent strings are concatenated.

3.5 Base64 Data

Raw binary data can be expressed in the Base64 format. As shown in Figure 14, Base64 data consists of a sequence of characters composed from the set {`A`–`Z`, `a`–`z`, `0`–`9`, `+`, `/`}. Each of the 64 characters in the encoding set corresponds to the 6-bit value assigned to it in Table 3.

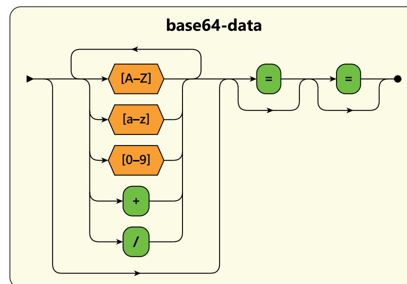

Figure 14. Base64 data is composed of uppercase and lowercase letters, numbers, the plus symbol, and the slash symbol. There may be up to two equal signs for padding at the end.

The number of encoded characters in a block of Base64 data must be 0, 2, or 3 modulo 4. Each group of four characters corresponds to exactly three decoded bytes having values determined as follows.

- The value of the first byte is given by the six bits encoded by the first character concatenated with the two most significant bits encoded by the second character.
- The value of the second byte is given by the four least significant bits encoded by the second character concatenated with the four most significant bits encoded by the third character.
- The value of the third byte is given by the two least significant bits encoded by the third character concatenated with the six bits encoded by the fourth character.

Char	Value	Char	Value	Char	Value	Char	Value
A	0	Q	16	g	32	w	48
B	1	R	17	h	33	x	49
C	2	S	18	i	34	y	50
D	3	T	19	j	35	z	51
E	4	U	20	k	36	0	52
F	5	V	21	l	37	1	53
G	6	W	22	m	38	2	54
H	7	X	23	n	39	3	55
I	8	Y	24	o	40	4	56
J	9	Z	25	p	41	5	57
K	10	a	26	q	42	6	58
L	11	b	27	r	43	7	59
M	12	c	28	s	44	8	60
N	13	d	29	t	45	9	61
O	14	e	30	u	46	+	62
P	15	f	31	v	47	/	63

Table 3. These are the 64 character values used in Base64 data.

If the number of encoded characters is 2 modulo 4, then the final two characters produce a single byte of decoded data, and the four least significant bits encoded by the second character are discarded. In this case, the encoded Base64 data may end with two equals sign characters as padding to make the total number of encoded characters a multiple of four. This padding is not required, and it is ignored if it is present.

If the number of encoded characters is 3 modulo 4, then the final three characters produce two bytes of decoded data, and the two least significant bits encoded by the third character are discarded. In this case, the encoded Base64 data may end with one equals sign character as padding to make the total number of encoded characters a multiple of four. This padding is not required, and it is ignored if it is present.

Whitespace may appear anywhere inside Base64 data, and it is ignored. Because the forward slash character has a specific meaning in the Base64 format, comments are not permitted to occur inside Base64 data.

4 Identifiers

An *identifier* is a sequence of characters composed from the set {A–Z, a–z, 0–9, _}, as shown in Figure 15. That is, an identifier is composed of uppercase and lowercase roman letters, the numbers 0 through 9, and the underscore. An identifier cannot begin with a number.

Identifiers are used to specify structure types, names, properties, and data states. The identifiers used for the 16 primitive data types listed in Table 1 are reserved as structure types, but they can still be used as names, properties, and data states.

All identifiers consisting of a single lowercase letter followed by zero or more numerical digits are reserved as structure types for future use by the language. A derivative format may define any other identifier as the type of a derived structure.

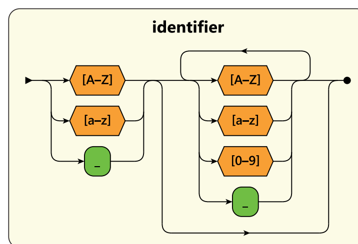

Figure 15. An identifier is composed of uppercase and lowercase roman letters, the numbers 0 through 9, and the underscore.

5 Names

Any structure may have a *name*. Names are used to identify specific structures so they can be referenced from within primitive structures or through property values. A name can be a global name or a local name. Each global name must be unique among all global names used inside the file containing it, and each local name must be unique among all local names used by its siblings in the structure tree. Local names can be reused inside different structures, and they can duplicate global names.

As shown in Figure 16, a name is composed of either a dollar sign character (`$`) or percent sign character (`%`) followed by an identifier with no intervening whitespace. A name that begins with a dollar sign is a global name, and a name that begins with a percent sign is a local name. A name is assigned to a structure by placing it immediately after the structure identifier (and no whitespace is technically required before the dollar sign). [*Example* —

```
Vertex $apex
{
    float {1.0, 2.0, 3.0}
}
```

The `Vertex` structure has the global name `$apex`. This structure can be referenced from elsewhere in the file by using the name `$apex` as a value of the `ref` type. — *End example*]

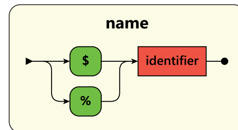

Figure 16. A name is composed of either a dollar sign character (`$`) or a percent sign character (`%`) followed by an identifier with no intervening whitespace.

6 References

A *reference* is a value that forms a link to a specific structure within an OpenDDL file. If the target structure has a global name, then the value of a reference to it is simply the name of the structure, beginning with the dollar sign character. If the target structure has a local name, then the value of a reference to it depends on the scope in which the reference appears. If the reference appears inside a structure that is a sibling of the target structure, then its value is the name of the target structure, beginning with the percent sign character. Otherwise, the value of the reference consists of a sequence of names, as shown in Figure 17, that identify a sequence of structures along a branch in a tree of structures. Only the first name in the sequence can be a global name, and the rest must be local names.

The value of a reference can also be keyword `null` to indicate that a reference has no target structure.

[*Example* — In the following code, the structure types `Person`, `Name`, and `Friends` are defined by a derivative format. References are used to link people to the data structures representing their friends.

```
Person $charles
{
    Name {string {"Charles"}}
    Friends {ref {$alice, $bob}}
}

Person $alice {...}
Person $bob {...}
```

— *End example*]

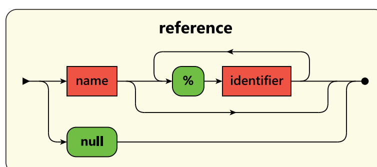

Figure 17. A reference is either the name of a structure or the keyword `null`. A structure may be identified by a sequence of names providing the path to the target along a branch in a tree of structures.

7 Primitive Data

Primitive structures contain homogeneous data of a single primitive data type in one of three possible forms.

7.1 Flat Data

The data contained in a primitive structure may consist of a flat, comma-separated list individual literal values, as shown in Figure 18. The size of the list is unbounded. In this case, the structure identifier is not followed by brackets, but only an optional name and the data itself enclosed in braces.

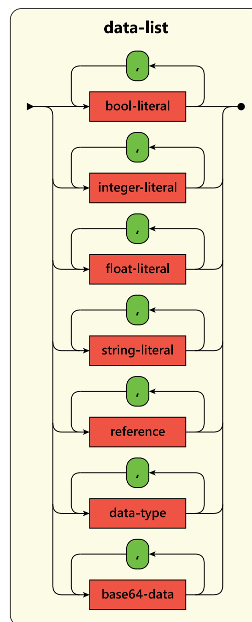

Figure 18. The data payload of a primitive structure may be a homogeneous list of literal values separated by commas.

Note that an implementation would use its knowledge of the primitive structure's data type to choose only a single rule in Figure 18, as opposed to allowing any of the types of data to appear inside the braces. (It is also not possible to disambiguate among the numerical data types without some extra information.)

7.2 Subarray Data

The data contained in a primitive structure may also be specified as a comma-separated list of *subarrays* of literal values, as shown in Figure 19. The size of the subarrays is specified by placing a positive

integer value inside brackets immediately following the structure identifier, preceding the structure's optional name. The data belonging to each subarray is then specified as a comma-separated list of values enclosed in braces. In this case, the identifiers shown in Figure 19 do not apply and may not appear in the structure's data.

As before, an implementation would choose only a single rule in Figure 19 based on the type of the primitive structure.

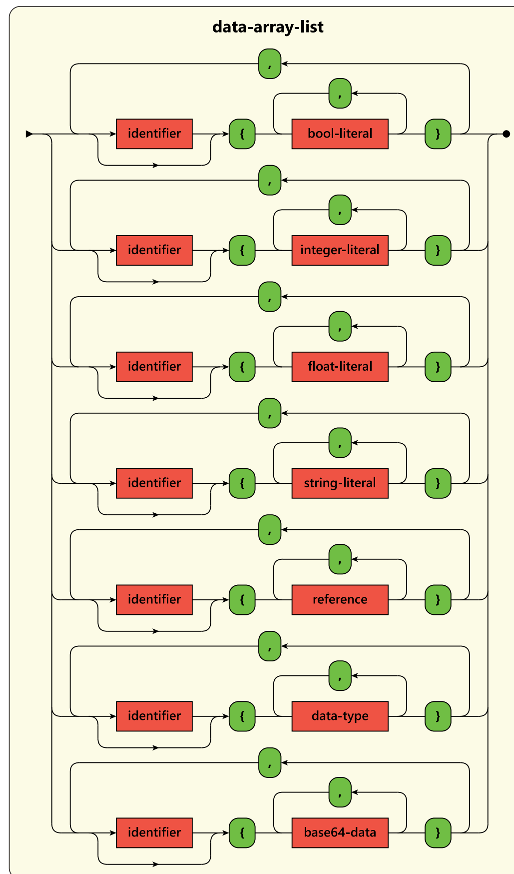

Figure 19. A data payload may consist of a list of subarrays separated by commas. Each subarray contains a homogeneous array of values enclosed in braces.

The number of elements in each subarray must always match the array size specified inside the brackets following the primitive type identifier. If the array size is one, then the braces are still required. While the size of the subarrays is fixed, the total number of subarrays is unbounded.

[*Example* — Suppose that a `VertexArray` structure expects to contain an array of 3D positions, each of which is specified as an array of three floating-point values. This would be written as follows.

```
VertexArray
{
    float[3]
    {
        {1.0, 2.0, 3.0}, {0.5, 0.0, 0.5}, {0.0, -1.0, 4.0}
    }
}
```

— *End example*]

7.3 Data States

Primitive structures containing subarrays may also specify *data states* that associate a particular state with each subarray. The meaning of data states are defined by a derivative format. The presence of data states is indicated by writing an asterisk (*) immediately after the subarray size enclosed in brackets. In this case, a state identifier may precede any subarray in the data payload. If a state identifier is omitted for any particular subarray, then the state associated with the preceding subarray continues to apply. The initial state is defined by the derivative format.

[*Example* — The following `Path` structure provides an example in which data states are used to specify how individual points in a list are to be interpreted. In this example, the state `M` causes the drawing position to be moved to the associated point, the state `L` causes a line to be drawn to the associated point, and the state `C` causes a cubic Bézier curve to be drawn using the preceding point and the following three points.

```
Path
{
    float[2]*
    {
        M{1.0, 1.0}, L{2.0, 1.0}, C{3.0, 1.0}, {3.0, 2.0}, {2.0, 3.0}
    }
}
```

— *End example*]

8 Properties

A derived structure may accept one or more *properties* that can be specified separately from the data contained inside the structure. Properties are written in a comma-separated list inside parentheses following the name of the structure, or just following the structure identifier if there is no name. As shown in Figure 20, each property is composed of a property identifier followed by an equals sign character (=) and the literal value of the property. The type of the property's value must be specified by some external source of information such as a schema or the implementation of the derivative format. For example, a string cannot be specified for a property that was expecting an integer. The specified type determines which subrule in Figure 20 is applied, and a mismatch must be detected at the time that the property is parsed.

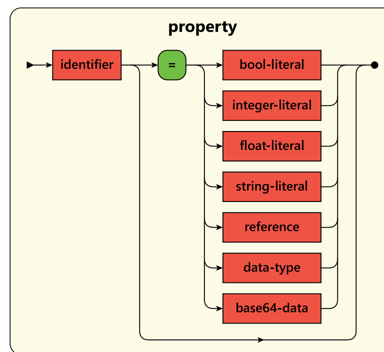

Figure 20. A property is composed of an identifier followed by an equals character (=) and the value of the property.

[*Example* — Suppose that a data structure called `Mesh` accepts a property called `lod` that takes an integer representing the level of detail to which it pertains. This property would be specified as follows.

```
Mesh (lod = 2)
{
    ...
}
```

If another property called `part` existed and accepted a string (perhaps to identify a body part), then that property could be added to the list as follows.

```
Mesh (lod = 2, part = "Left Hand")
{
    ...
}
```

— *End example*]

The order in which properties are listed is not significant. Derivative formats may require that certain properties always be specified. Optional properties must always have a default value or be specially handled as being in an unspecified state. The same property can be specified more than once in the same property list, and in such a case, all but the final value specified for the same property must be ignored.

Boolean properties allow a special syntax in which the assignment of a value of `true` or `false` can be omitted. In this case, the presence of the property implies that its value is `true`. This is a useful shorthand notation for Boolean properties having a default value of `false`. Properties having any other type must include an assigned value.

A structure is allowed to have properties that are not recognized by the implementation of a derivative format in order to support extensions. Unrecognized properties must be ignored and must not generate an error. However, the value assigned to such a property must still be parsable as one of the primitive data types.

The syntax does not allow primitive structures to have a property list.

9 Comments and Whitespace

The language supports C++-style block comments and single-line comments as follows:

- Any occurrence of `/*` begins a comment that ends immediately after the next occurrence of `*/`. Such comments do not nest.
- Any occurrence of `//` begins a comment that ends immediately after the next newline character.

If any sequence `/*`, `*/`, or `//` appears inside a character literal or string literal, then it is part of the literal value and not treated as a comment.

Comments may include any Unicode characters encoded as UTF-8. The only other place where non-ASCII characters are allowed is inside a string literal (see Section 3.4).

Comments cannot occur inside base64 data (see Section 3.5).

All characters having a value in the range [1, 32] (which includes the space, tab, newline, and carriage return characters), as well as all characters belonging to comments, are considered to be whitespace in OpenDDL. Any arbitrarily long contiguous sequence of whitespace characters is equivalent to a single space character.

10 Formal Grammar

For reference, the formal grammar defining the OpenDDL syntax using Backus-Naur Form and regular expressions is shown in Listing 1. The figures displayed throughout this specification precisely correspond to this grammar. A syntactically valid OpenDDL file satisfies the `file` rule at the end of the listing.

Listing 1. This is the formal grammar defining the OpenDDL syntax.

```
identifier         ::= [A-Za-z_] [0-9A-Za-z_]*

name               ::= ("$" | "%") identifier

reference          ::= name ("%" identifier)* | "null"

hex-digit          ::= [0-9A-Fa-f]

escape-char        ::= '\"' | "\'" | "\?" | "\\" | "\a" | "\b" | "\f"
                     | "\n" | "\r" | "\t" | "\v"
                     | "\x" hex-digit hex-digit

bool-literal       ::= "false" | "0" | "true" | "1"

decimal-literal    ::= [0-9] ("_"? [0-9])*

hex-literal        ::= ("0x" | "0X") hex-digit ("_"? hex-digit)*

octal-literal      ::= ("0o" | "0O") [0-7] ("_"? [0-7])*

binary-literal     ::= ("0b" | "0B") ("0" | "1") ("_"? ("0" | "1"))*

char-literal       ::= "'" ([#x20-#x26#x28-#x5B#x5D-#x7E]
                       | escape-char)+ "'"

integer-literal    ::= ("+" | "-")? (decimal-literal | hex-literal
                       | octal-literal | binary-literal | char-literal)

float-literal      ::= ("+" | "-")?
                       (([0-9] ("_"? [0-9])* ("." ([0-9] ("_"? [0-9])*)?)?
                       | "." [0-9] ("_"? [0-9])*)
                       (("e" | "E") ("+" | "-")? [0-9] ("_"? [0-9])*)?
                       | hex-literal | octal-literal | binary-literal)

string-literal     ::= ('"' ([#x20-#x21#x23-#x5B#x5D-#x7E#xA0-#xD7FF#xE000-
                       #xFFFD#x010000-#x10FFFF] | escape-char
                       | "\u" hex-digit hex-digit hex-digit hex-digit
```

```
                      | "\U" hex-digit hex-digit hex-digit hex-digit
                      hex-digit hex-digit)* '"')+

data-type          ::= "bool" | "b" | "int8" | "i8" | "int16" | "i16"
                    | "int32" | "i32" | "int64" | "i64" | "uint8" | "u8"
                    | "uint16" | "u16" | "uint32" | "u32" | "uint64" | "u64"
                    | "half" | "h" | "float" | "f" | "double" | "d" | "float16"
                    | "f16" | "float32" | "f32" | "float64" | "f64" | "string"
                    | "s" | "ref" | "r" | "type" | "t" | "base64" | "z"

base64-data        ::= [A-Za-z0-9]* "="? "="?

data-list          ::= bool-literal ("," bool-literal)*
                    | integer-literal ("," integer-literal)*
                    | float-literal ("," float-literal)*
                    | string-literal ("," string-literal)*
                    | reference ("," reference)*
                    | data-type ("," data-type)*
                    | base64-data ("," base64-data)*

data-array-list    ::= identifier? "{" bool-literal (","
                      bool-literal)* "}" ("," identifier? "{"
                      bool-literal ("," bool-literal)* "}")*
                    | identifier? "{" integer-literal (","
                      integer-literal)* "}" ("," identifier? "{"
                      integer-literal ("," integer-literal)* "}")*
                    | identifier? "{" float-literal (","
                      float-literal)* "}" ("," identifier? "{"
                      float-literal ("," float-literal)* "}")*
                    | identifier? "{" string-literal (","
                      string-literal)* "}" ("," identifier? "{"
                      string-literal ("," string-literal)* "}")*
                    | identifier? "{" reference (","
                      reference)* "}" ("," identifier? "{"
                      reference ("," reference)* "}")*
                    | identifier? "{" data-type (","
                      data-type)* "}" ("," identifier? "{"
                      data-type ("," data-type)* "}")*
                    | identifier? "{" base64-data (","
                      base64-data)* "}" ("," identifier? "{"
                      base64-data ("," base64-data)* "}")*

property           ::= identifier ("=" (bool-literal | integer-literal
                      | float-literal | string-literal | reference
                      | data-type | base64-data))?

structure          ::= data-type (name? "{" data-list? "}"
                      | "[" integer-literal "]" "*"? name?
```

```
                        "{" data-array-list? "}")
                      | identifier name? ("(" (property ("," property)*)?
                        ")")? "{" structure* "}"

file                 ::= structure*
```

11 Revision History

Version 3.0

The following changes were made in OpenDDL version 3.0.

- The ability to specify `0` and `1` as literal boolean values was added.
- The option to omit the assignment of `true` to a boolean property was added.
- State identifiers were added for subarray data.
- The Base64 data type was added.

Version 2.0

The following changes were made in OpenDDL version 2.0.

- The short primitive data type identifiers `b`, `i8`, `i16`, `i32`, `i64`, `u8`, `u16`, `u32`, `u64`, `h`, `f`, `d`, `s`, `r`, and `t` were added as alternatives to the long identifiers.
- The `float16`, `float32`, and `float64` primitive data type identifiers and their short forms `f16`, `f32`, and `f64` were added as alternatives to the identifiers `half`, `float`, and `double`.
- All structure identifiers consisting of a single lowercase letter followed by zero or more numerical digits were reserved for future use by the language.

Version 1.1

The following changes were made in OpenDDL version 1.1.

- The `half` primitive data type was added to accommodate 16-bit floating-point numbers.
- The ability to use underscore characters as visual separators in numerical literals was added.
- Support for octal literals was added.