Ion 1.1 Specification

Ion Team

Contents

Chapter 1

Introduction

Draft Status

This document is currently a working draft and subject to change.

Audience

This documents presents the formal specification for the Ion 1.1 data format. This document is not intended to be used as a user guide or as a cook book, but as a reference to the syntax and semantics of the Ion data format and its logical data model.

Chapter 2

What's New in Ion 1.1

We will go through a high-level overview of what is new and different in Ion 1.1 from Ion 1.0 from an implementer's perspective.

2.1 Motivation

Ion 1.1 has been designed to address some of the trade-offs in Ion 1.0 to make it suitable for a wider range of applications. Ion 1.1 now makes length prefixing of containers optional, and makes the interning of symbolic tokens optional as well. This allows for applications that write data more than they read data or are constrained by the writer in some way to have more flexibility. Data density is another motivation. Certain encodings (e.g., timestamps, integers) have been made more compact and efficient, but more significantly, macros now enable applications to have very flexible interning of their data's structure. In aggregate, data transcoded from Ion 1.0 to Ion 1.1 should be more compact.

2.2 Backwards Compatibility

Ion 1.1 is backwards compatible to Ion 1.0. Backwards compatibility is defined as being able to *parse* Ion 1.0 encoded data **and** ensuring that any data model values produced by Ion 1.1 that are *not* system values must be representable in Ion 1.0. To wit, any data that can be produced and read by an application in Ion 1.1 must have an equivalent representation in Ion 1.0.

Important
Discussion: Is this statement too weak? Specifically, should we be attempting to "fill in the holes" in the Ion data model around system values? Should we require that Ion 1.1 implementations *produce* Ion 1.0 data?

Ion 1.1 is **not** required to preserve Ion 1.0 binary encodings in Ion 1.1 encoding contexts (i.e., the type codes and lower-level encodings are not preserved in the new version). The Ion Version Marker (IVM) is used to denote the different versions of the syntax. Ion 1.1 does retain text compatibility with Ion 1.0 in that the changes are a strict superset of the grammar, however due to the updated system symbol table, symbol IDs referred to using the `$n` syntax for symbols beyond the 1.0 system symbol table are not compatible.

2.3 Text Syntax Changes

Ion 1.1 text **must** use the `$ion_1_1` version marker at the top-level of the data stream or document.

The only syntax change for the text format is the introduction of **encoding expression** (**E-expression**) syntax, which allows for the invocation of macros in the data stream. This syntax is grammatically similar to S-expressions, except that these expressions are opened with `(:` and closed with `)`. For example, `(:a 1 2)` would expand the macro named `a` with the arguments `1` and `2`. See the Macros, Templates, and Encoding-Expressions section for details.

This syntax is allowed anywhere an Ion value is allowed:

Figure 1. E-expression Examples

```
// At the top level
(:foo 1 2)

// Nested in a list
[1, 2, (:bar 3 4)]

// Nested in an S-expression
(cons a (:baz b))

// Nested in a struct
{c: (:bop d)}
```

E-expressions are also grammatically allowed in the field name position of a struct and when used there, indicate that the expression should expand to a struct value that is merged into the enclosing struct:

Figure 2. E-Expression in field position of struct.

```
{
    a:1,
    b:2,
    (:foo 1 2),
    c: 3,
}
```

In the above example, the E-expression `(:foo 1 2)` must evaluate into a struct that will be merged between the `b` field and the `c` field. If it does not evaluate to a struct, then the above is an error.

2.4 Binary Encoding Changes

Ion 1.1 binary encoding reorganizes the type descriptors to support compact E-expressions, make certain encodings more compact, and certain lower priority encodings marginally less compact. The IVM for this encoding is the octet sequence `0xE0 0x01 0x01 0xEA`.

2.4.1 Inlined Symbolic Tokens

Important
Discussion: Should we call this something else (e.g., *non-interned*)?

In binary Ion 1.0, symbol values, field names, and annotations are required to be encoded using a symbol ID in the local symbol table. For some use cases (e.g., as write-once, read-maybe logs) this creates a burden on the writer and may not actually be efficient for an application. Ion 1.1 introduces optional binary syntax for encoding inline UTF-8 sequences for these tokens which can allow an encoder to have flexibility in whether and when to add a given symbolic token to the symbol table.

Ion text requires no change for this feature as it already had inline symbolic tokens without using the local symbol table. Ion text also has compatible syntax for representing the local symbol table and encoding of symbolic tokens with their position in the table (i.e., the `$id` syntax).

2.4.2 Delimited Containers

In Ion 1.0, all data is length prefixed. While this is good for optimizing the reading of data, it requires an Ion encoder to buffer any data in memory to calculate the data's length. Ion 1.1 introduces optional binary syntax to allow containers to be encoded with an end marker instead of a length prefix.

2.4.3 Low-level Binary Encoding Changes

Ion 1.0's `VarUInt` and `VarInt` encoding primitives used big-endian byte order and used the high bit of each byte to indicate whether it was the final byte in the encoding. `VarInt` used an additional bit in the first byte to represent the integer's sign. Ion 1.1 replaces these primitives with more optimized versions called `FlexUInt` and `FlexInt`.

`FlexUInt` and `FlexInt` use little-endian byte order, avoiding the need for reordering on x86 architectures. Rather than using a bit in each byte to indicate the width of the encoding, `FlexUInt` and `FlexInt` front-load the continuation bits. In most cases, this means that these bits all fit in the first byte of the representation, allowing a reader to determine the complete size of the encoding without having to inspect each byte individually. Finally, `FlexInt` does not use a separate bit to indicate its value's sign. Instead, it uses two's complement representation, allowing it to share much of the same structure and parsing logic as its unsigned counterpart. Benchmarks have shown that in aggregate, these encoding changes are between 1.25 and 3x faster than Ion 1.0's `VarUInt` and `VarInt` encodings depending on the host architecture.

Ion 1.1 supplants Ion 1.0's `Int` encoding primitive with a new encoding called `FixedInt`, which uses two's complement notation instead of sign-and-magnitude. A corresponding `FixedUInt` primitive has also been introduced; its encoding is the same as Ion 1.0's `UInt` primitive.

A new primitive encoding type, `FlexSym`, has been introduced to flexibly encode symbol IDs and symbolic tokens with inline text.

2.4.4 Type Encoding Changes

All Ion types use the new low-level encodings as specified in the previous section. Many of the opcodes used in Ion 1.0 have been re-organized primarily to make E-expressions compact.

Typed `null` values are now encoded in two bytes using the `0xEB` opcode.

Lists and S-expressions have two encodings: a length-prefixed encoding and a new delimited form that ends with the `0xF0` opcode.

Struct values have three encodings: a length-prefixed encoding which uses symbol IDs for its field names, a length-prefixed encoding which uses `FlexSym` for its field names (allowing for inline symbol text as needed), and a delimited form which encodes its field names with `FlexSym` and ends with an escape (`0x00`) followed by the `0xF0` opcode. (There is no delimited form with symbol ID field names).

Symbol values have two encodings: one is the Ion 1.0-style encoding using the symbol ID, and the other one is structurally identical to the encoding of strings, supplying its text's UTF-8 bytes inline.

Annotation sequences are a prefix to the value they decorate, and no longer have an outer length container. They are now encoded with an opcode that specifies a single annotation with value following, an opcode that specifies two annotations with a value following, and finally, an opcode that specifies a variable length of annotations followed by a value. The latter encoding is similar to how Ion 1.0 annotations are encoded with the exception that there is no outer length.

Important
Discussion: Should we provide an op-code for length prefixing the entire annotation? If so, where should it go? E.g, make the variable length SID based annotations support this.

Integers now use a `FixedInt` sub-field instead of the Ion 1.0 encoding which used sign-and-magnitude (with two opcodes).

Decimals are structurally identical to their Ion 1.0 counterpart with the exception of the negative zero coefficient. The Ion 1.1 `FlexInt` encoding is two's complement, so negative zero cannot be encoded directly with it. Instead, an encoding opcode is allocated specifically for encoding decimals with a negative zero coefficient.

Timestamps no longer encode their sub-field components as octet-aligned fields. The Ion 1.1 format uses a packed bit encoding and has a biased form (encoding the year field as an offset from 1970) to make common encodings of timestamp easily fit in a 64-bit word for microsecond and nanosecond precision (with UTC offset or unknown UTC offset). Benchmarks have shown this new encoding to be 59% faster to encode and 21% faster to decode. A non-biased, arbitrary length timestamp with packed bit encoding is defined for uncommon cases.

2.4.5 Encoding Expressions in Binary

E-expressions in binary are encoded with an opcode that encodes the *macro identifier* or an opcode that specifies a `FlexUInt` for the macro identifier. This is followed by the encoding of the arguments to the E-expression. The macro's definition statically determines how the arguments are to be laid out. An argument may be a full Ion value with encoding opcode, or it could be a lower-level encoding (e.g., fixed width integer or `FlexInt`/`FlexUInt`).

2.5 Macros, Templates, and Encoding-Expressions

Ion 1.1 introduces a new kind of encoding called **encoding expression** (**E-expression**). These expressions are (in text syntax) similar to S-expressions, but they are not part of the data model and are *evaluated* into one or more Ion values (called a *stream*) which enable compact representation of Ion data. E-expressions represent the invocation of either system defined or user defined **macros** with arguments that are either themselves E-expressions, value literals, or container constructors (list, sexp, struct syntax containing E-expressions) corresponding to the formal parameters of the macro's definition. The resulting stream is then expanded into the resulting Ion data model.

At the top level, the stream becomes individual top-level values. Consider for illustrative purposes an E-expression `(:values 1 2 3)` that evaluates to the stream `1`, `2`, `3` and `(:void)` that evaluates to the empty stream. In the following examples, `values` and `void` are the names of the macros being invoked and each line is equivalent.

Figure 3. Top-level E-expressions

```
a (:values 1 2 3) b (:void) c
a 1 2 3 b c
```

Within a list or S-expression, the stream becomes additional child elements in the collection.

Figure 4. E-expressions in lists

```
[a, (:values 1 2 3), b, (:void), c]
[a, 1, 2, 3, b, c]
```

Figure 5. E-expressions in S-expressions

```
(a (:values 1 2 3) b (:void) c)
(a 1 2 3 b c)
```

Within a struct at the field name position, the resulting stream must contain structs and each of the fields in those structs become fields in the enclosing struct (the value portion is not specified); at the value position, the resulting stream of values becomes fields with whatever field name corresponded before the E-expression (empty stream elides the field all together). In the following examples, let us define `(:make_struct c 5)` that evaluates to a single struct `{c: 5}`.

Figure 6. E-expressions in structs

```
{a: (:values 1 2 3), b: 4, (:make_struct c 5), d: 6, e: (:void)}
{a: 1, a: 2, a: 3, b: 4, c: 5, d: 6}
```


2.5.1 Encoding Context and Modules

In Ion 1.0, there is a single *encoding context* which is the local symbol table. In Ion 1.1, the *encoding context* becomes the following:

- The local symbol table which is a list of strings. This is used to encode/decode symbolic tokens.
- The local macro table which is a list of macros. This is used to reference macros that can be invoked by E-expressions.
- A mapping of a string name to **module** which is an organizational unit of symbol definitions and macro definitions. Within the encoding context, this name is unique and used to address a module's contents either as the list of symbols to install into the local symbol table, the list of macros to install into the local macro table, or to qualify the name of a macro in a text E-expression or the definition of a macro.

The **module** is a new concept in Ion 1.1. It contains:

- A list of strings representing the symbol table of the module.
- A list of macro definitions.

Modules can be imported from the catalog (they subsume shared symbol tables), but can also be defined locally. Modules are referenced as a group to allocate entries in the local symbol table and local macro table (e.g., the local symbol table is initially, implicitly allocated with the symbols in the `$ion` module).

Ion 1.1 introduces a new system value (an *encoding directive*) for the encoding context (see the ***TBD*** section for details.)

Figure 7. Ion encoding directive example

```
$ion_encoding::{
  modules:         [ /* module declarations - including imports */ ],
  install_symbols: [ /* names of declared modules */ ],
  install_macros:  [ /* names of declared modules */ ]
}
```

Important
This is still being actively worked and is provisional.

2.5.2 Macro Definitions

Macros can be defined by a user either directly in a local module within an encoding directive or in a shared module defined externally (i.e., shared module). A macro has a name which must be unique in a module **or** it may have no name.

Ion 1.1 defines a list of *system macros* that are built-in in the module named `$ion`. Unlike the system symbol table, which is always installed and accessible in the local symbol table, the system macros are both always accessible to E-expressions and not installed in the local macro table by default (unlike the local symbol table).

In Ion binary, macros are always addressed in E-expressions by the offset in the local macro table. System macros may be addressed by the system macro identifier using a specific encoding op-code. In Ion text, macros may be addressed by the offset in the local macro table (mirroring binary), its name if its name is unambiguous within the local encoding context, or by qualifying the macro name/offset with the module name in the encoding context. An E-expression can *only* refer to macros installed in the local macro table or a macro from the system module. In text, an E-expression referring to a system macro that **is not** installed in the local macro table, must use a qualified name with the `$ion` module name.

For illustrative purposes let's consider the module named `foo` that has a macro named `bar` at offset 5 installed at the begining of the local macro table.

Figure 8. E-expressions name resolution in text

```
// allowed if there are no other macros named 'bar'
(:bar)
// fully qualified by module--always allowed
(:foo:bar)
// by local macro table offset
(:5)
// system macros are always addressable by name--in binary this would be a different offset ↩
     with a different opcode
(:$ion:void)
```

2.5.3 Macro Definition Language

User defined macros are defined by their parameters and **template** which defines how they are invoked and what stream of data they evaluate to. This template is defined using a domain specific Ion macro definition language with S-expressions. A template defines a list of zero or more parameters that it can accept. These parameters each have their own cardinality of expression arguments which can be specified as *exactly one*, *zero or one*, *zero or more*, and *one or more*. Furthermore the template defines what type of argument can be accepted by each of these parameters:

- Specific type(s) of Ion value.
- Lower-level binary data (e.g. fixed width integers or `VarUInt`) for efficient encodings of the E-expressions in binary.
- Specific *macro shaped arguments* to allow for structural composition of macros and efficient encoding in binary.

The macro definition includes a **template body** that defines how the macro is expanded (see the ***TBD*** section for details). In the language, system macros, macros defined in previously defined modules in the encoding context, and macros defined previously in the current module are accessible to be invoked with `(name ...)` syntax where `name` is the macro to be invoked. Certain names in the expression syntax are reserved for special forms (i.e., `quote`, `if`, `when`, `unless`, and `each`). When a macro name is shadowed by a special form, or is ambiguous with respect to all macros visible, it can always be qualified with `(':module:name' ...)` syntax where `module` is the name of the module and `name` is the offset or name of the macro. Referring to a previously defined macro name *within* a module may be qualified with `(':name' ...)` syntax.

INFORMATION: ***TBD*** put an easy to access example of a macro definition.

2.5.4 Shared Modules

Ion 1.1 extends the concept of *shared symbol table* to be a *shared module*. An Ion 1.0 shared symbol table is a shared module with no macro definitions. A new schema for the convention of serializing shared modules in Ion are introduced in Ion 1.1 (see the ***TBD*** section for details). An Ion 1.1 implementation should support containing Ion 1.0 shared symbol tables and Ion 1.1 shared modules in its catalog.

2.6 System Symbol Table Changes

The system symbol table in Ion 1.1 adds the following symbols:

ID	Symbol Text
10	`$ion_encoding`
11	`$ion_literal`

System macro identifiers are namespaced separately and therefore do not have entries in the system symbol table.

Important
These assignments are provisional. Specifically assignments for the macro definition language have not been established.

2.7 E-Expression Calling Conventions in Binary

Important
WIP: This section is incomplete and needs rework.

An E-expression specifies the macro ID, followed by the macro's arguments. The macro's *parameter list* determines which how these arguments are laid out. When all parameters for a macro have *exactly one* argument, each argument is encoded using their normal Ion binary encodings.

When a parameter to a macro may have multiple argument expressions (i.e., *zero or one*, *one or more*, or *zero or more*), a bit stream aligned to the nearest byte in big endian order precedes the encoded values/invocations to indicate the presence or absence of the argument at that position. This bit stream is only used when one or more such parameters with low-level encoding (tagless) *or* two or more parameters with typed opcode (tagged) encoding exist.

For each parameter that is specified to have a *zero or more* or *one or more* cardinality, its argument prefixed with a `VarInt` that specifies the length of the argument:

- When *positive* this is an *octet length* prefix for the values/invocations.
- When *negative* this is a *count* for the values/invocations. * When *zero* **and** the encoding of the arguments use a full encoding opcode per argument the arguments are delimited by the `0xAD` (end indicator).
- When *zero* **and** the encoding of the arguments use lower-level encodings, this denotes empty arguments.

This `VarInt` is not required when an E-expression encoding has the argument bit-stream indicating no argument is present (i.e., empty).

Chapter 3

Macros by Example

Before getting into the technical details of Ion's macro and module system, it will help to be more familiar with the *use* of macros. We'll step through increasingly sophisticated use cases, some admittedly synthetic for illustrative purposes, with the intent of teaching the core concepts and moving parts without getting into the weeds of more formal specification.

Ion macros are defined using a domain-specific language that is in turn expressed via the Ion data model. That is, macro definitions are Ion data, and use Ion features like S-expressions and symbols to represent code in a LISP-like fashion. In this document, the fundamental construct we explore is the *macro definition*, denoted using an S-expression of the form `(macro name ...)` where **`macro`** is a keyword and *`name`* must be a symbol denoting the macro's name.

> **Note**
> S-expressions of that shape only declare macros when they occur in the context of an encoding module, which is the topic of a chapter to come. We will completely ignore modules for now, and the examples below omit this context to keep things simple.

3.1 Constants

The most basic macro is a constant:

```
(macro pi []
  3.141592653589793)
```

This declaration defines a macro named `pi`. The `[]` is the macro's *signature*, in this case a trivial one that declares no parameters. The `3.141592653589793` is a similarly trivial *template*, an expression in Ion 1.1's domain-specific language for defining macro functions. This macro accepts no arguments and always returns a constant value.

To use `pi` in an Ion document, we write an *encoding expression* or *E-expression*:

```
$ion_1_1
(:pi)
```

The syntax `(:pi)` looks a lot like an S-expression. It's not, though, since colons cannot appear unquoted in that context. Ion 1.1 makes use of syntax that is not valid in Ion 1.0—specifically, the `(:` digraph—to denote E-expressions. Those characters must be followed by a reference to a macro, and we say that the E-expression is an invocation of the macro. Here, `(:pi)` is an invocation of the macro named `pi`.

> **Note**
> We also call these "smile expressions" when we're feeling particularly casual.

That document is equivalent to the following, in the sense that they denote the same data:

```
$ion_1_1
3.141592653589793
```

The process by which the Ion implementation turns the former document into the latter is called *macro expansion* or just *expansion*. This happens transparently to Ion-consuming applications: the stream of values in both cases are the same. The documents have the same content, encoded in two different ways. It's reasonable to think of `(:pi)` as a custom encoding for `3.141592653589793`, and the notation's similarity to S-expressions leads us to the term "encoding expression".

> **Note**
> Any Ion 1.1 document with macros can be fully-expanded into an equivalent Ion 1.0 document.

We can streamline future examples with a couple conventions. First, assume that any E-expression is occurring within an Ion 1.1 document; second, we use the relation notation, ⇒, to mean "expands to". So we can say:

```
(:pi) ⇒ 3.141592653589793
```

3.2 Simple Templates

Most macros are not constant, they accept inputs that determine their results.

```
(macro price
  [a, c]                          // signature
  { amount: a, currency: c })     // template
```

This macro has a signature that declares two parameters, named `a` and `c`, and it therefore accepts two arguments when invoked.

```
(:price 99 USD) ⇒ { amount: 99, currency: USD }
```

> **Note**
> We are careful to distinguish between the views from "inside" and "outside" the macro: *parameters* are the names used by a macro's implementation to refer to its expansion-time inputs, while *arguments* are the data provided to a macro at the point of invocation. In other words, we have "formal" parameters and "actual" arguments.

The struct in this macro is our first non-trivial *template*, an expression in Ion's new domain-specific language for defining macro functions. This expression language treats Ion scalar values (except for symbols) as literals, giving the decimal in `pi`'s template its intended meaning. Expressions that are structs are interpreted *almost* literally: the field names are literal, but the field "values" are arbitrary expressions. This is why the `amount` and `currency` field names show up as-is in the expansion. We call these almost-literal forms *quasi-literals*.

The template language also treats lists quasi-literally, and every element inside the list is an expression. Here's a silly macro to illustrate:

```
(macro reverse [a, b] [b, a])
```

```
(:reverse first 1990) ⇒ [1990, first]
```

The sub-expressions in these templates demonstrate that the expression language treats symbols as variable references. The symbols in the templates above (`a` and `c` in `price`; `a` and `b` in `reverse`) refer to the parameters of their respective surrounding macros, and during expansion they are "filled in" with the values supplied by the invocation of the macro.

These names are part of the macro language that have no relation to data encoded using the macro:


```
(:reverse c {amount:a, currency:c}) ⇒ [{amount:a, currency:c}, c]
```

Symbols in an E-expression are *not* part of the expression language and do not reference macro parameters or any other named entity. From the point of view of `reverse`'s template, the inputs are literal data.

E-expressions can nest, so we could also encode the same data using `price`:

```
(:reverse first (:price a c))
  ⇒ (:reverse first {amount:a, currency:c})
  ⇒ [{amount:a, currency:c}, first]
```

As the example suggests, expansion steps proceed "inside out" and the outer macro receives the results from the inner invocation.

3.3 Invoking Macros from Templates

Template expressions that are S-expressions are *operator invocations*, where the operators are either macros or *special forms*. We start with the former:

```
(macro website_url
  [path]
  (make_string "https://www.amazon.com/" path))
```

In this case, the S-expression `(make_string ...)` is an invocation of the system macro (that is, a built-in function) `make_string`, which concatenates its arguments to produce a single string:

```
(:website_url "gp/cart") ⇒ "https://www.amazon.com/gp/cart"
```

In the template language, macro invocations can appear almost anywhere:

```
(macro detail_page_url
  [asin]
  (website_url (make_string "dp/" asin)))
```

```
(:detail_page_url "B08KTZ8249") ⇒ "https://www.amazon.com/dp/B08KTZ8249"
```

Note
While this doesn't look like much of an improvement, the full string takes 38 bytes to encode, but the macro invocation takes as few as 12 bytes.

Careful readers will note that templates can use `[...]` and `{...}` notation to construct lists and structs, but `(...)` doesn't construct S-expressions. This gap is filled by the built-in macro `make_sexp` which accepts any number of arguments and puts them in a sexp:

```
(macro double_sexp [val] (make_sexp val val))
```

```
(:make_sexp true 19.3 null) ⇒ (true 19.3 null)
(:double_sexp double) ⇒ (double double)
```

3.3.1 E-expressions Versus S-expressions

We've now seen two ways to invoke macros, and their difference deserves thorough exploration.

An E-expression is an encoding artifact of a serialized Ion document. It has no intrinsic meaning other than the fact that it represents a macro invocation. The meaning of the document can only be determined by expanding the macro, passing the E-expression's arguments to the function defined by the macro. This all happens as the Ion document is parsed, transparent to the reader of the document. In casual terms, E-expressions are expanded away before the application sees the data.

Within the template-expression language, you can define new macros in terms of other macros, and those invocations are written as S-expressions. Unlike E-expressions, these are normal Ion data structures, consumed by the Ion system and interpreted as code. Further, they only exist in the context of a macro definition, inside an encoding module, while E-expressions can occur *anywhere* in an Ion document.

> **Warning**
> It's entirely possible to write a macro that can generate all or part of a macro definition. We don't recommend that you spend time considering such things at this point.

These two invocation forms are syntactically aligned in their calling convention, but are distinct in context and "immediacy". E-expressions occur anywhere and are invoked immediately, as they are parsed. S-expression invocations occur only within macro definitions, and are only invoked if and when that code path is ever executed by invocation of the surrounding macro.

3.4 Special Form: `literal`

When a template-expression is syntactically an S-expression, its first element must be a symbol that matches either a set of keywords denoting the special forms, or the name of a previously-defined macro. The interpretation of the S-expression's remaining elements depends on how the symbol resolves. In the case of macro invocations, we've seen above that the following elements are (so far!) arbitrary template expressions, but for special forms that's not always the case. The **`literal`** form makes this clear:

```
(macro USD_price [dollars] (price dollars (literal USD)))
```

```
(:USD_price 12.99) ⇒ { amount: 12.99, currency: USD }
```

In this template, we can't just write `(price dollars USD)` because the symbol `USD` would be treated as an unbound variable reference and a syntax error, so we turn it into literal data by "escaping" it with **`literal`**.

> **Tip**
> Our documents use bold typewriter face to distinguish special forms and keywords from symbols referencing macros and parameters.

The critical point is that special forms are "special" precisely because they cannot be expressed as macros and must therefore receive bespoke syntactic treatment. Since the elements of macro-invocation expressions are themselves expressions, when you want something to *not* be evaluated that way, it must be a special form.

Finally, these special forms are part of the template language itself, and are not visible to encoded data: the E-expression `(:literal foo)` must necessarily refer to some user-defined macro named `literal`, not to this special form. As an aside, there is no need for such a form in E-expressions, because in that context symbols and S-expressions are not "evaluated", and everything is literal except for E-expressions (which are not data, but encoding artifacts).

> **Note**
> Ion 1.1 defines a number of built-in macros and special forms. While this document covers the highlights, it is not a complete reference to all features.

3.5 Parameter Types

In our examples so far, the macro signatures have been simple lists of parameter names, and each parameter accepts a value of any type. But this is often undesirable, since the resulting output could violate the intended schema or the macro-expansion could fail in hard-to-diagnose ways:

```
(:detail_page_url [true]) ⇒ error: make_string expects a string
```

This E-expression cannot be expanded because `make_string` requires its arguments to be textual values, and `[true]` is not a string or symbol. But this failure happens within the implementation of `detail_page_url`, not the point where the error occurred. In this example, those points are only one step removed, but it's not hard to imagine macros where the call stack is deep enough to make diagnosis difficult.

To detect problems close to their source, macro signatures can declare type constraints on their parameters:

```
(macro detail_page_url
  [(asin string)]
  (website_url (make_string "dp/" asin)))
```

This example reveals additional syntax for parameter declarations. So far, a parameter was declared by a symbol denoting its name, now we have an S-expression containing a name and a type. Here the parameter's name is `asin`, its type is `string`. The intended input domain is now clear and the Ion parser can emit an error sooner:

```
(:detail_page_url [true]) ⇒ error: detail_page_url expects a string
```

In this context the types include all the normal "concrete" Ion types, abstract supertypes like **`number`**, **`text`**, and **`lob`**, and the unconstrained "top type" **`any`**. The latter is the default type, and the signature `[foo]` is equivalent to `[(foo any)]` meaning that the parameter `foo` accepts one value of any type.

Tip
These types also serve a second purpose: they can allow the binary encoding to be more compact by avoiding type tags or using fixed-width values.

3.6 Rest Parameters

Sometimes we want a macro to accept an arbitrary number of arguments, in particular *all the rest of them*. The `make_string` macro is one of those, concatenating all of its arguments into a single string:

```
(:make_string)                ⇒ ""
(:make_string "a")            ⇒ "a"
(:make_string "a" "b"    )    ⇒ "ab"
(:make_string "a" "b" "c")    ⇒ "abc"
(:make_string "a" "b" "c" "d") ⇒ "abcd"
```

To make this work, the definition of make_string is effectively:

```
(macro make_string [(parts text ...)] ...)
```

This says that `parts` is a *rest parameter* accepting zero or more arguments of type **`text`**. The **`...`** modifier can only occur on the last parameter, declaring that "all the rest" of the arguments will be passed to that one name.

Note
The Ion grammar treats identifiers like `text` and operators like **`...`** as separate tokens regardless of whether they are separated by whitespace. We think it's easier to read without whitespace and will use that convention from now on.

At this point our distinction between parameters and arguments becomes apparent, since they are no longer one-to-one: this macro with one parameter can be invoked with one argument, or twenty, or none. We describe the acceptable number of values for a parameter as its *cardinality*. In the examples so far, all parameters have had *exactly-one* cardinality, while `parts` has *zero-or-more* cardinality. We'll see additional cardinalities soon!

Tip
To declare a rest parameter that requires at least one value, use the `...+` modifier.

3.7 Arguments and Results are Streams

The inputs to and results from a macro are modeled as streams of values. When a macro is invoked, each argument produces a stream of values, and within the macro definition, each parameter name refers to the corresponding stream, not to a specific value. The declared cardinality of a parameter constrains the number of elements produced by its stream, and is verified by the macro expansion system.

More generally, the results of all template expressions are streams. While most expressions produce a single value, various macros and special forms can produce zero or more values.

We have everything we need to illustrate this, via another system macro, `values`:

```
(macro values [(vals any...)] vals)
```

```
(:values 1)            ⇒ 1
(:values 1 true null) ⇒ 1 true null
(:values)              ⇒ nothing
```

The `values` macro accepts any number of arguments and returns their values, effectively a multi-value identity function. We can use this to explore how streams combine in E-expressions.

3.7.1 Splicing in Encoded Data

When an E-expression occurs at top-level or within a list or S-expression, the results are spliced into the surrounding container:

```
[first, (:values), last]           ⇒ [first, last]
[first, (:values "middle"), last] ⇒ [first, "middle", last]
(first (:values left right) last) ⇒ (first left right last)
```

This also applies wherever a tagged type can appear inside an E-expression:

```
(first (:values (:values left right) (:values)) last) ⇒ (first left right last)
```

Note that each argument-expression always maps to one parameter, even when that expression returns too-few or too-many values.

```
(macro reverse [(a any), (b any)]
  [b, a])
```

```
(:reverse (:values 5 USD))    ⇒ error: 'reverse' expects 2 arguments, given 1
(:reverse 5 (:values) USD)    ⇒ error: 'reverse' expects 2 arguments, given 3
(:reverse (:values 5 6) USD) ⇒ error: argument 'a' expects 1 value, given 2
```

In this example, the parameters expect exactly one argument, producing exactly one value. When the cardinality allows multiple values, then the argument result-streams are concatenated. We saw this (rather subtly) above in the nested use of `values`, but can also illustrate using the rest-parameter to `make_string`, which we'll expand here in steps:

```
(:make_string (:values) a (:values b (:values c) d) e)
  ⇒ (:make_string a (:values b (:values c) d) e)
  ⇒ (:make_string a (:values b c d) e)
  ⇒ (:make_string a b c d e)
  ⇒ "abcde"
```

Splicing within sequences is straightforward, but structs are trickier due to their key/value nature. When used in field-value position, each result from a macro is bound to the field-name independently, leading to the field being repeated or even absent:

```
{ name: (:values) }          ⇒ { }
{ name: (:values v) }        ⇒ { name: v }
{ name: (:values v ann::w) } ⇒ { name: v, name: ann::w }
```

An E-expression can even be used in place of a key-value pair, in which case it must return structs, which are merged into the surrounding container:

```
{ a:1, (:values), z:3 }             ⇒ { a:1, z:3 }
{ a:1, (:values {}), z:3 }          ⇒ { a:1, z:3 }
{ a:1, (:values {b:2}), z:3 }       ⇒ { a:1, b:2, z:3 }
{ a:1, (:values {b:2} {z:3}), z:3 } ⇒ { a:1, b:2, z:3, z:3 }

{ a:1, (:values key "value") } ⇒ error: struct expected for splicing into struct
```

3.7.2 Splicing in Template Expressions

The preceding examples demonstrate splicing of E-expressions into encoded data, but similar stream-splicing occurs within the template language, making it trivial to convert a stream to a list:

```
(macro int_list
  [(vals int...)]
  [ vals ])
(macro clumsy_bag
  [(elts any...)]
  { '': elts })
```

```
(:int_list)   ⇒ []
(:clumsy_bag) ⇒ {}

(:int_list 1 2 3)    ⇒ [1, 2, 3]
(:clumsy_bag true 2) ⇒ {'':true, '':2}
```

Streams and lists are different, there's no flattening involved, and declared types are verified:

```
(:int_list 1 [2] 3) ⇒ error: [2] is not an int
```

TODO: demonstrate splicing in TDL macro invocations

3.8 Mapping Templates Over Streams: `for`

Another way to produce a stream is via a mapping form. The **`for`** special form evaluates a template once for each value provided by a stream or streams. Each time, a local variable is created and bound to the next value on the stream.

```
(macro prices
  [(currency symbol), (amounts number...)]
  (for [(amt amounts)]                        ❶
    (price amt currency)))
```

1. The first subform of **for** is a list of binding pairs, S-expressions containing a variable names and a template expressions. Here, that template expression is simply a parameter reference, so each individual value from the `amounts` is bound to the name `amt` before the `price` invocation is expanded.

```
(:prices GBP 10 9.99 12.)
  ⇒ {amount:10, currency:GBP} {amount:9.99, currency:GBP} {amount:12., currency:GBP}
```

More than one stream can be iterated in parallel, and iteration terminates when any stream becomes empty.

```
(macro zip [(front any*), (back any*)]   ❶
  (for [(f front),
        (b back)]
    [f, b]))
```

1. The `*` means that the parameter accepts any number of values; see Section 3.10.3.

```
(:zip (:values 1 2 3) (:values a b))
  ⇒ [1, a] [2, b]
```

Note
This termination rule is under discussion; see https://github.com/amazon-ion/ion-docs/issues/201

3.9 Empty Streams: `void`

The empty stream is an important edge case that requires careful handling and communication. We'll use the term *void* to mean "empty stream". We'll even mint the word *voidable* to describe parameters that can accept empty streams, like the `...`s above.

Correspondingly, the built-in macro `void` accepts no values and produces an empty stream:

```
(:int_list (:void)) ⇒ []
(:int_list 1 (:void) 2) ⇒ [1, 2]
[(:void)]    ⇒ []
{a:(:void)} ⇒ {}
```

When used as a macro argument, a `void` invocation (like any other expression) counts as one argument:

```
(:pi (:void)) ⇒ error: 'pi' expects 0 arguments, given 1
```

The special-case E-expression `(:)` is synonymous with `(:void)` and is useful as a more succinct expression of absent arguments:

```
(:int_list (:)) ⇒ []
(:int_list 1 (:) 2) ⇒ [1, 2]
```

Tip
While `void` and `values` both produce the empty stream, the former is preferred for clarity of intent and terminology.

3.10 Cardinality

As described earlier, parameters are all streams of values, but the number of values can be controlled by the parameter's cardinality. So far we have seen the default exactly-one and the `...` (zero-or-more) cardinality modifiers, and in total there are six:

Modifier	Cardinality
`!`	exactly-one value
`?`	zero-or-one value
`+`	one-or-more values
`*`	zero-or-more values
`...`	zero-or-more values, as "rest" arguments
`...+`	one-or-more values, as "rest" arguments

3.10.1 Exactly-One

Many parameters expect exactly one value and thus have *exactly-one cardinality*. This is the default for ungrouped parameters, but the `!` modifier can be used for clarity.

This cardinality means that the parameter requires a stream producing a single value, so one might refer to them as *singleton streams* or just *singletons* colloquially.

3.10.2 Zero-or-One

A parameter with the modifier `?` has *zero-or-one cardinality*, which is much like exactly-one cardinality, except the parameter is voidable. That is, it accepts an empty-stream argument as a way to denote an absent parameter.

```
(macro temperature
  [(degrees decimal), (scale symbol?)]
  {degrees: degrees, scale: scale})
```

Since the scale is voidable, we can pass it void:

```
(:temperature 96 F)    ⇒ {degrees:96, scale:F}
(:temperature 283 (:)) ⇒ {degrees:283}
```

Note that the result's `scale` field has disappeared because no value was provided. It would be more useful to fill in a default value, and to do that we introduce a special form that can detect void:

```
(macro temperature
  [(degrees decimal), (scale symbol?)]
  {degrees: degrees, scale: (if_void scale (literal K) scale)})
```

```
(:temperature 96 F)    ⇒ {degrees:96,  scale:F}
(:temperature 283 (:)) ⇒ {degrees:283, scale:K}
```

The `if_void` form is if/then/else syntax testing stream emptiness. It has three sub-expressions, the first being a stream to check. If and only if that stream is void (it produces no values), the second sub-expression is expanded and its results are returned by the `if_void` expression. Otherwise, the third sub-expression is expanded and returned.

Note
Exactly one branch is expanded, because otherwise the void stream might be used in a context that requires a value, resulting in an errant expansion error.

To refine things a bit further, trailing voidable arguments can be omitted entirely:

```
(:temperature 283) ⇒ {degrees:283, scale:K}
```

3.10.3 Zero-or-More

A parameter with the modifier `*` has *zero-or-more cardinality*. This modifier behaves the same as `...` from the perspective of its template, but it can be used in any position, not just last place.

```
(macro prices
  [(amount number*), (currency symbol)]
  (for [(amt amount)]
    (price amt currency)))
```

The calling convention for `*` is different from `...` since the "all the rest" convention can't be used to draw the boundaries of the stream. Instead, we need a single expression that produces the desired values:

```
(:prices (:) JPY)              ⇒ void
(:prices 54 CAD)               ⇒ {amount:54, currency:CAD}
(:prices (:values 10 9.99) GBP) ⇒ {amount:10, currency:GBP} {amount:9.99, currency:GBP}
```

3.10.4 One-or-More

A parameter with the modifier `+` has *one-or-more cardinality*, which works like `*` except the resulting stream must produce at least one value. To continue using our `prices` example:

```
(macro prices
  [(amount number+), (currency symbol)]
  (for [(amt amount)]
    (price amt currency)))
```

```
(:prices (:) JPY) ⇒ error: at least one value expected for + parameter
(:prices 54 CAD)               ⇒ {amount:54, currency:CAD}
(:prices (:values 10 9.99) GBP) ⇒ {amount:10, currency:GBP} {amount:9.99, currency:GBP}
```

A macro's final parameter can use a variant of rest parameters with one-or-more cardinality, denoted by the `...+` modifier:

```
(macro thanks [(names text...+)]
  (make_string "Thank you to my Patreon supporters:\n"
    (for [(n names)]
      (make_string "  * " n "\n"))))
```

```
(:thanks) ⇒ error: at least one value expected for ...+ parameter

(:thanks Larry Curly Moe) ⇒
'''\
Thank you to my Patreon supporters:
  * Larry
  * Curly
  * Moe
'''
```

3.11 Grouped Parameters

The non-rest versions of multi-value parameters can be annoying to invoke, since they require the use of `values` or some other template to produce the stream of values. To streamline invocation, a macro can opt-in to special syntax that uses a list as delimiting syntax to group the applicable sub-expressions. This is denoted by wrapping the parameter's type in `[]`:


```
(macro prices
  [(amount [number]),       ❶
   (currency symbol)]
  (for [(amt amount)]
    (price amt currency)))
```

❶ Note the use of `[]` around **`number`**.

This is referred to as a *grouped parameter*, and at invocation it requires a list delimiting its *argument group*:

```
(:prices [1, 2, 3] GBP) ⇒ {amount:1, currency:GBP}
                          {amount:2, currency:GBP}
                          {amount:3, currency:GBP}
```

Within the group, the invocation can have any number of arguments, including macro invocations. The macro parameter produces the results of those expressions, concatenated into a single stream, and the expander verifies that each value on that stream is acceptable by the parameter's declared type.

```
(:prices [1, (:values 2 3), 4] GBP) ⇒ {amount:1, currency:GBP}
                                      {amount:2, currency:GBP}
                                      {amount:3, currency:GBP}
                                      {amount:4, currency:GBP}
```

Important
To avoid ambiguity, the delimiter is required even for singleton values. Consider this macro:

```
(macro ouch [(stuff [list])] ...)
```

Without this rule, the E-expression `(:ouch [])` would be ambiguous whether the parameter was intended to be void or a singleton empty-list value.

Grouping says whether multiple *arguments* can be provided, while cardinality describes the number of *values* those argument(s) must produce. The parameter declaration `(amount [number])` makes grouping explicit, with a default cardinality of zero-or-more. The declaration `(amount [number]+)` is also valid, indicating that the sequence of arguments must produce at least one value.

Tip
Grouped parameters cannot use the **`?`** and **`!`** modifiers; there's no point in requiring a grouping list when no more than one value is allowed.

Tip
Rest parameters are effectively another grouping mode, so they cannot be combined with `[]`.

Delimiting sequences and `values` expressions may appear similar because they both denote streams of values, but they are not interchangeable:

```
(:prices (:values 10 9.99 12.) GBP) ⇒ error: delimiting list or sexp expected
(:prices (:) GBP)                   ⇒ error: delimiting list or sexp expected
```

That's because the binary representation of these parameters uses a tagless format for these delimiters to keep the common case as dense as possible. It's not possible to replace that container with a macro invocation, and the text form mirrors that limitation. If the parameter type allows (see Section 3.13), you can call a macro inside the delimiter, with no loss of generality:

```
(:prices [(:values 10)] GBP) ⇒ {amount:10, currency:GBP}
```

3.12 Optional Arguments

When a trailing parameter is voidable, an invocation can omit its corresponding arguments or group, as long as no following parameter is being given an argument or group. We've seen this as applied to `...` rest-parameters, but it also applies to **?** and ***** parameters, with or without groups:

```
(macro optionals
  [(a [any]), (b any?), (c any!), (d [any]), (e any?), (f any...)]
  (make_list a b c d e f))
```

Since `d`, `e`, and `f` are all voidable, they can be omitted by invokers. But `c` is required so `a` and `b` must always be present, at least as an empty group:

```
(:optionals [] (:) "value for c") ⇒ ["value for c"]
```

Now `c` receives the symbol `for_c` while the other parameters are all void. If we want to provide just `e`, then we must also provide a group for `d`:

```
(:optionals [] (:) "value for c" [] "value for e")
  ⇒ ["value for c", "value for e"]
```

3.13 Tagless and Fixed-Width Types

In Ion 1.0, the binary encoding of every value starts off with a "type tag", an opcode that indicates the data-type of the next value and thus the interpretation of the following octets of data. In general, these tags also indicate whether the value has annotations, and whether it's null.

These tags are necessary because the Ion data model allows values of any type to be used anywhere. Ion documents are not schema-constrained: nothing forces any part of the data to have a specific type or shape. We call Ion "self-describing" precisely because each value self-describes its type via a type tag.

If schema constraints are enforced through some mechanism outside the serializer/deserializer, the type tags are unnecessary and may add up to a non-trivial amount of wasted space. when you observe that the overhead for each value also includes length information: encoding an octet of data takes two octets on the stream.

Ion 1.1 tries to mitigate this overhead in the binary format by allowing macro parameters to use more-constrained *primitive types*. These are subtypes of the concrete types, constrained such that type tags are not necessary in the binary form. In general this can shave 4-6 bits off each value, which can add up in aggregate. In the extreme, that octet of data can be encoded with no overhead at all.

The following primitive types are available:

Primitive Type	Description
var_symbol	Tagless symbol (SID or text)
var_string	Tagless string
var_int	Tagless, variable-width signed int
var_uint	Tagless, variable-width unsigned int
int8 int16 int32 int64	Fixed-width signed int
uint8 uint16 uint32 uint64	Fixed-width unsigned int
float16 float32 float64	Fixed-width float

To define a tagless parameter, just declare one of the primitive types:

```
(macro point
  [(x var_int), (y var_int)]
  {x: x, y: y})
```

```
(:point 3 17) ⇒ {x:3, y:17}
```

The type constraint has no real benefit here in text, as primitive types aim to improve the binary encoding. TODO talk about binary length improvement.

This density comes at the cost of flexibility. Primitive types cannot be annotated or null, and arguments cannot be expressed using macros, like we've done before:

```
(:point null.int 17)   ⇒ error: primitive var_int does not accept nulls
(:point a::3 17)       ⇒ error: primitive var_int does not accept annotations
(:point (:values 1) 2) ⇒ error: cannot use macro for a primitive argument
```

While Ion text syntax doesn't use tags—the types are built into the syntax—these errors ensure that a text E-expression may only express things that can also be expressed using an equivalent binary E-expression.

For the same reasons, a parameter accepting more than one tagless argument can only be expressed by grouped or rest parameters, not by ungrouped forms. For example, `(v var_int+)` and `(v int32*)` are not accepted.

A subset of the tagless types are *fixed-width*: they are binary-encoded with no per-value overhead.

```
(macro byte_array
  [(bytes uint8...)]
  [bytes])
```

Invocations of this macro are encoded as a sequence of untagged octets, because the macro definition constrains the argument shape such that nothing else is acceptable. A text invocation is written using normal ints:

```
(:byte_array 0 1 2 3 4 5 6 7 8) ⇒ [0, 1, 2, 3, 4, 5, 6, 7, 8]
(:byte_array 9 -10 11)          ⇒ error: -10 is not a valid uint8
(:byte_array 256)               ⇒ error: 256 is not a valid uint8
```

As above, Ion text doesn't have syntax specifically denoting "8-bit unsigned integers", so to keep text and binary capabilities aligned, the parser rejects invocations where an argument value exceeds the range of the binary-only type.

Primitive types have inherent tradeoffs and require careful consideration, but in the right circumstances the density wins can be significant.

3.14 Macro Shapes

We can now introduce the final kind of input constraint, macro-shaped parameters. To understand the motivation, consider modeling a scatter-plot as a list of points:

```
[{x:3, y:17}, {x:395, y:23}, {x:15, y:48}, {x:2023, y:5}, ...]
```

Lists like these exhibit a lot of repetition. Since we already have a `point` macro, we can eliminate a fair amount:

```
[(:point 3 17), (:point 395 23), (:point 15 48), (:point 2023 5), ...]
```

This eliminates all the `x`s and `y`s, but leaves repeated macro invocations. We can try to wrap this in another macro, but we find the type constraints insufficient, since the tightest we can go is **`struct`**, and things aren't really any better:

```
(macro scatterplot [(points struct...)]
  [points])
```

```
(:scatterplot (:point 3 17) (:point 395 23) (:point 15 48) (:point 2023 5) ...)
```

What we'd like is to eliminate the `point` calls and just write a stream of pairs, something like:

```
(:scatterplot (3 17) (395 23) (15 48) (2023 5) ...)
```

We can achieve exactly that with a macro-shaped parameter, in which we use the `point` macro as a pseudo-type:

```
(macro scatterplot [(points point...)]   ❶
  [points])
```

❶ `point` is not one of the built-in types, so its a reference to the macro of that name defined earlier.

```
(:scatterplot (3 17) (395 23) (15 48) (2023 5) ...)
  ⇒
  [{x:3, y:17}, {x:395, y:23}, {x:15, y:48}, {x:2023, y:5}, ...]
```

Each argument S-expression like `(3 17)` is *implicitly an E-expression* invoking the `point` macro. The argument mirrors the shape of the inner macro, without repeating its name. Further, expansion of the implied `points` happens automatically, so the overall behavior is just like the preceding struct-based variant and the `points` parameter produces a stream of structs.

The binary encoding of macro-shaped parameters are similarly tagless, eliding any opcodes mentioning `point` and just writing its arguments with minimal delimiting.

Macro types can be grouped and/or combined with cardinality modifiers, following the same rules as tagless types. Note that grouped macro types require callers to use two layers of delimiting containers: and outer list for the group, and an inner S-expression for each macro instance:

```
(macro scatterplot
  [(points [point] +), (x_label string), (y_label string)]
  { points: [points], x_label: x_label, y_label: y_label })
```

```
(:scatterplot [(3 17), (395 23), (15 48), (2023 5)] "hour" "widgets")
  ⇒
  {
    points: [{x:3, y:17}, {x:395, y:23}, {x:15, y:48}, {x:2023, y:5}],
    x_label: "hour",
    y_label: "widgets"
  }
```

As with non-macro arguments, you cannot replace a grouping list with a macro invocation. Further, you can't use a macro invocation as an *element* of the delimiting-list:

```
(:scatterplot (:make_points 3 17 395 23 15 48 2023 5) "hour" "widgets")
  ⇒ error: delimiting list or sexp expected, found :make_points

(:scatterplot [(3 17), (:make_points 395 23 15 48), (2023 5)] "hour" "widgets")
  ⇒ error: sexp expected with args for 'point', found :make_points

(:scatterplot [(3 17), (:point 395 23), (15 48), (2023 5)] "hour" "widgets")
  ⇒ error: sexp expected with args for 'point', found :point
```

This limitation mirrors the binary encoding, where both the delimiting list and the individual macro invocations are tagless and there's no way to express a macro invocation.

> **Tip**
> The primary goal of macro-shaped arguments, and tagless types in general, is to increase density by tightly constraining the inputs.

3.15 Return Types

TODO

Chapter 4

Modules by Example

The prior chapter explored macro definitions while ignoring the contexts within which those definitions exist. This chapter covers that context top-down.

4.1 Ion 1.0 Encoding Environment

An Ion *document* is a stream of octets conforming to either the Ion text or binary specification. (For our purposes here, a document does not necessarily exist as a file, and isn't necessarily finite.) The interpretation of those octets is guided by an *encoding environment*, the context maintained by an Ion implementation while encoding or decoding a document. The Ion 1.0 encoding environment is just the local symbol table.

The encoding environment is controlled by *directives* embedded in the document at top-level. These are encoding artifacts and are not part of the application data model.

Ion 1.0 has two forms of directives:

- An Ion Version Marker (IVM) resets the environment to the default provided by that version of Ion.
- An `$ion_symbol_table` struct defines a new environment that takes effect immediately after the struct closes.

A *segment* is a contiguous portion of a document that uses the same encoding environment. Segment boundaries are caused by directives: an IVM starts a new segment, while an `$ion_symbol_table` struct ends a segment, defining a new one that starting immediately afterwards. As a result, non-IVM directives are always encoded using the environment of the segment that contains them.

TODO Ion text docs always start with a 1.0 segment until an IVM is encountered.

4.2 Modules from the Outside

In Ion 1.1, you define, share, and install symbols and macros using *encoding modules*. The logical interface to a module has three main components: a spec version, a symbol table, and a macro table.

A module's *spec version* indicates which Ion specification it uses. This ensures the module has stable semantics over time. A module can only be used in segments encoded with that version or later.

> **Important**
> **Discussion**: The above may be too strict; use solely for symbols could be more relaxed.

A module's *exported symbol table* is simply a sequence of strings. These denote the text of symbols, and are equivalent in meaning to the `symbols` list of an Ion 1.0 shared symbol table.

A module's *exported macro table* is a sequence of <name, macro> pairs. Names can be null, in which case the corresponding macro can be referenced by its zero-based index in the table, known as its *exported address*. Non-null names in the table must all be unique, so that a name-to-macro mapping function is well-defined.

Tip
Macros have their own identity independent of the names that map to them. It's possible for the same macro to have multiple addresses and/or names.

To reuse macros across documents, *shared modules* subsume the capabilities of shared symbol tables while remaining backwards-compatible with their current schema and catalog semantics.

Important
All existing Ion shared symbol tables **are** encoding modules. Such modules only declare symbols and not macros.

4.3 Ion 1.1 Encoding Environment

In Ion 1.1, the encoding environment includes:

- The current Ion version, because a document may have segments using different Ion versions.
- The *available modules*, a name to module mapping.
- The current symbol table, assembled from a subset of the available modules.
- The current macro table, assembled from a subset of the available modules.

Note
In Ion 1.0, the local symbol table *is* the encoding environment.

Upon encountering the `$ion_1_1` IVM, the environment is reset to the default state, in which:

- The Ion version is 1.1.
- The available modules contains only the `$ion` module, version 2 (v1 being Ion 1.0).
- The macro table is empty.
- The symbol table is the Ion 1.1 system symbol table.

To customize this environment, we use an *encoding directive*: a top-level S-expression annotated with `$ion_encoding`. Like `$ion_symbol_table`, this directive defines a new encoding environment that goes into effect immediately after the directive closes.

Note
We use the term "encoding directive" to refer to the `$ion_encoding` S-expression, and "local symbol table directive" to refer to the `$ion_symbol_table` struct. Both forms are valid in Ion 1.1.

The general syntax of an encoding directive is as follows:

```
$ion_encoding::(
  (retain ...)        // Reuse selected modules from the current segment
  (load ...)          // Get a shared module from the catalog
  (module ...)        // Define a new module inline
  ...
  (symbol_table ...)  // Install modules into the symbol table
  (macro_table ...)   // Install modules into the macro table
)
```

Each syntactic form affects one of the main components of the environment. The **symbol_table** and **macro_table** clauses specify the layout of those tables, while the preceding clauses enumerate the available modules that may be installed into them.

Note
Using an S-expression instead of a struct constrains the order in which clauses are encountered, making it both more code-like and easier to parse.

Let's look at some examples illustrating the relation between `$ion_symbol_table` and `$ion_encoding`.

4.4 Defining Local Symbols

The most basic Ion encoding scenario uses only locally-defined symbols. In Ion 1.0, this is expressed as follows:

```
$ion_1_0
$ion_symbol_table::{
  symbols: ["s1", "s2"]
}
```

Here's an Ion 1.1 document that's equivalent, in the sense that it allocates symbol IDs in the same order. (The IDs will be different, though, due to new system symbols.)

```
$ion_1_1
$ion_encoding::(
  (module extracted
    (symbol_table [ "s1", "s2" ]))
  (symbol_table extracted)
)
```

The definition of the local symbol table has been refactored into two parts. First, the list of symbols is expressed inside a module named `extracted`. Then, the symbols from that module are installed to form the new local symbol table. Compared to the behavior of `$ion_symbol_table`, this is akin to defining a named symbol table "inline" to hold local symbols, then defining the local symbol table only via `imports` and no `symbols` field.

Let's look more closely at the definition of `extracted`:

```
(module extracted
  (symbol_table [ "s1", "s2" ]))
```

The **module** keyword starts an S-expression that defines a new *inline module* with the given name. The **symbol_table** keyword starts a subform that defines the module's exported symbol table. This clause accepts a list of strings, using the same syntax and semantics as the `symbols` field of `$ion_shared_symbol_table`.

Once this module is defined, we can install its symbols into the directive's symbol table:

```
  (symbol_table extracted)
```

This clause accepts a series of symbols that match names declared in the `modules` field. The resulting local symbol table is simply the concatenation of the exported symbol tables of those modules. This works the same way as the `imports` field of `$ion_symbol_table`.

4.5 Importing Symbols

Given the equivalencies above, we could perform a naive round-trip of the preceding 1.1 document back to 1.0. First, turn the `extracted` module into the equivalent shared symbol table:

```
$ion_shared_symbol_table::{
  name: "com.example.extracted",
  version: 1,
  symbols: ["s1", "s2"]
}
```

Then translate `(symbol_table extracted)` into its 1.0 equivalent:

```
$ion_1_0
$ion_symbol_table::{
  imports: [{ name: "com.example.extracted", version: 1, max_id: 2 }]
}
```

Note
Even ignoring Ion 1.1, this is how you would extract local symbols into a new shared symbol table.

The latter imports-only document has this 1.1 equivalent:

```
$ion_1_1
$ion_encoding::(
  (load extracted "com.example.extracted" 1 2)
  (symbol_table extracted)
)
```

Here we see a new form inside the `modules` field that imports a module into the encoding environment and assigns it a name. The **`load`** keyword starts an S-expression that expects three or four arguments. The first is a symbolic name that we can use later to refer to the imported module. The remaining arguments are effectively the `name`, `version` and `max_id` fields of the 1.0 `imports` struct, with only the max_id being optional in this form.

Tip
From the perspective of Ion 1.1, shared symbol tables *are* encoding modules.

4.6 Declaring Multiple Modules

Let's look at a scenario with both imported and locally-defined symbols:

```
$ion_1_0
$ion_symbol_table::{
  imports: [{ name: "com.example.shared1", version: 1, max_id: 10 },
            { name: "com.example.shared2", version: 2, max_id: 20 }],
  symbols: ["s1", "s2"]
}
```

Here's the Ion 1.1 equivalent in terms of symbol allocation order:

```
$ion_1_1
$ion_encoding::(
  (load m1 "com.example.shared1" 1 10)
  (load m2 "com.example.shared2" 2 20)
  (module local_syms (symbol_table ["s1", "s2"]))
  (symbol_table m1 m2 local_syms)
)
```

Just as in the 1.0 version, this allocates ten symbol IDs for `m1` (as requested by its max_id argument), twenty symbol IDs for `m2`, then the two locally-defined symbols.

By decoupling symbol-table importing from installation, Ion 1.1 allows some encoding techniques that are not possible in 1.0. For example, we can give local symbols smaller IDs than imported symbols by installing `local_syms` first:

```
$ion_1_1
$ion_encoding::(
  (load m1 "com.example.shared1" 1 10)
  (load m2 "com.example.shared2" 2 20)
  (module local_syms (symbol_table ["s1", "s2"]))
  (symbol_table local_syms m1 m2)                     // 'local_syms' is first
)
```

While there is little impact in this example, when imported tables are large this technique can ensure that local symbols fit into the first 256 addresses, using only two bytes to encode in binary.

4.7 Extending the Current Symbol Table

The last 1.0 feature to examine is adding symbols to the current symbol table:

```
$ion_1_0
$ion_symbol_table::{
  symbols: ["s1", "s2"]
}

// ... application data ...

$ion_symbol_table::{
  imports: $ion_symbol_table,
  symbols: ["s3", "s4"]
}
```

To achieve this in Ion 1.1, we must copy the available modules from the current segment into the next, while also defining a new module for the additional symbols.

```
$ion_1_1
$ion_encoding::(
  (module syms (symbol_table ["s1", "s2"]))
  (symbol_table syms)
)

// ... application data ...

$ion_encoding::(
  (retain *)
  (module syms2 (symbol_table ["s3", "s4"]))
  (symbol_table syms syms2)
)
```

The **`retain`** clause indicates that all (*) of the available modules in the current encoding environment are to be reused in the new one. Alternatively, individual modules can be named, if only a subset is desired.

Here again, Ion 1.1 enables a new technique: we can prepend new symbols to the current symbol table.

```
$ion_encoding::(
  (retain *)
  (module syms2 (symbol_table ["s3", "s4"]))
  (symbol_table syms2 syms)                     // 'syms2' is first
)
```


4.8 Installing and Using Macros

The local macro table works in essentially the same way as the local symbol table: you import or define modules that export macros, then you enumerate the modules whose macros you want to install. The lists of exported macros from each of those modules are concatenated to form a contiguous address space so that any macro can be referenced by an integer.

We can now define a small module for two-dimensional geometry, finally showing macro definitions in full context:

```
$ion_1_1
$ion_encoding::(
  (module geo
    (macro_table
      (macro point [(x int), (y int)]
        {x: x, y: y})
      (macro line  [(a point), (b point)]
        [a, b])))
  (macro_table geo)
)
(:point 17 28)
(:line (1 2) (3 4))
```

This `geo` module defines macros instead of symbols, using the **`macro`** definition syntax explored throughout Chapter 3.

The `macro_table` field works much like `symbol_table`: it assembles a macro table by concatenating the exported macro tables of the referenced modules, which must be declared within the adjacent `modules` field.

With macros installed, the document can then invoke them using E-expressions, and the `point` and `line` invocations above produce results equivalent to:

```
{x:17, y:28}
[{x:1, y:2}, {x:3, y:4}]
```

There are a couple differences between the local symbol and macro tables. In both cases, their entries can be addressed via offsets in the table, but the local macro table does not start with system macros so user-defined macros start at address zero. In the document above, the first macro in the first module is `point`, so we could write:

```
(:0 17 28) ⇒ {x:17, y:28}
```

Further, the local macro table tracks the names of installed modules, so that macros can be addressed using qualified names like `(:geo:point 17 28)`. Any ambiguity among exported macro names may be resolved at the point of reference using this syntax. Qualified addresses work as well, so `:geo:0` resolves to the macro at address 0 of module `geo`, which is `point`.

All told, Ion text offers four variants of macro references. Each of these lines is equivalent:

```
(:0         17 28)  (:1         (1 2) (3 4))
(:geo:0     17 28)  (:geo:1     (1 2) (3 4))
(:geo:point 17 28)  (:geo:line  (1 2) (3 4))
(:point     17 28)  (:line      (1 2) (3 4))
```

This topic is more interesting when more than one module is involved, so let’s table this for now.

4.9 Shared Modules

Macros are most useful when they’re shared across documents, and for that we use *shared modules*, a generalization of Ion 1.0’s shared symbol tables. As discussed in Section 4.2, they export both a symbol table and a macro table.

> **Tip**
> In Ion 1.1, a shared symbol table *is* a shared module.

NOTE: We intend to propose a new schema for shared modules, akin to the new `$ion_encoding` schema. That should be easier to explain and understand than the format below.

For backwards compatibility purposes, shared modules are expressed using the legacy schema for shared symbol tables, adding a `module` field to hold macro definitions:

```
$ion_1_0
$ion_shared_module::$ion_1_1::(
  (catalog_key "com.example.graphics.3d" 1)
  (symbol_table ["x", "y", "z"])
  (macro_table
    (macro point [(x int), (y int), (z int)]
      {x: x, y: y, z: z})
    (macro line  [(a point), (b point)]
      [a, b])
    (macro poly  [(first point), (second point), (rest point...+)]
      [first, second, rest]))
)
```

This S-expression is very similar to the **module** S-expression inside `$ion_encoding`. Here, no symbolic name is declared, since one will be assigned when the module is loaded. No **symbols** clause is allowed, since those are expected to be in the legacy `symbols` field. For comparison, here's a functionally-equivalent inline definition:

```
$ion_encoding::(
  (module g3d
    (symbol_table ["x", "y", "z"])
    (macro_table
      (macro point [(x int), (y int), (z int)]
        {x: x, y: y, z: z})
      (macro line  [(a point), (b point)]
        [a, b])
      (macro poly  [(first point), (second point), (rest point...+)]
        [first, second, rest])))
  ...
```

The `$ion_shared_module` document above is encoded in Ion 1.0 format, despite containing information that only applies to an Ion 1.1 implementation. Shared symbol tables are communicated via the Ion data model, which is guaranteed consistent across all Ion 1.x specifications, so encoding modules can be expressed using any Ion version with no change in semantics. To accomplish this, we require the IVM-like `$ion_1_1` annotation on the definition, denoting the spec version that provides meaning to the module.

4.10 Using Shared Macros

With a shared module at hand, we can load it and install its macros:

```
$ion_1_1
$ion_encoding::(
  (load g3d "com.example.graphics.3d" 1)  // Load it
  (macro_table g3d)                       // Install it
)
```

We can also combine shared and inline modules:

```
$ion_1_1
$ion_encoding::(
  (load g3d "com.example.graphics.3d" 1)
  (module geo
    (macro_table
      (macro point [(x int), (y int)]
        {x: x, y: y})
```



```
      (macro line  [(a point), (b point)]
        [a, b])))
  (macro_table geo g3d)
)
```

We now have a problem: the names `point` and `line` are ambiguous, referring to two different macros each. Thankfully, we can use qualified references to disambiguate:

```
(:geo:point 17 28)  (:g3d:point 20 18 45)
(:geo:0     17 28)  (:g3d:0     20 18 45) // Equivalent
```

In fact, we *must* do so. An E-expression with an un unqualified macro name is erroneous when the name is ambiguous, meaning that two installed modules map it to different macros.

```
(:point 17 28) ⇒ error: ':point' is ambiguous, exported by 'geo' and 'g3d'.
```

Another thing to note in the directive used above is that the **`load`** `g3d` declaration includes a symbol table name and version, but no max_id argument. As with imports in a local symbol table, absence of max_id forces the Ion implementation to acquire the symbol table entity with exactly the stated version. While this is generally not best-practice for importing symbols, exact-match is a **requirement** for using any macros in the module or installing it in a **`macro_table`**. In other words, when a document is encoded using macros, the Ion decoder will always use the *exact* version of those macros that was used when encoding the data.

Tip
With respect to macros, there is no assumption of compatibility across versions of modules.

4.11 Private Imports

In Ion 1.0, the ability to import symbols from a shared symbol table is limited to local symbol table; shared tables cannot be dynamically composed via `imports`. This isn't much of a problem in practice, since symbols are trivial to manage. Macros are more sophisticated entities, and most macros are implemented in terms of other macros. This makes it valuable to support transitive import of macros between shared modules.

Let's revisit our scatter plot example and build a module for expressing charts for various data sets. First we take our basic geometric macros and package them in a shared module:

```
$ion_shared_module::$ion_1_1::(
  (catalog_key "com.example.geometry" 1)
  (macro_table
    (macro point [(x int), (y int)]
      {x: x, y: y})
    (macro line  [(a point), (b point)]
      [a, b]))
)
```

Now we build another shared module using it:

```
$ion_shared_module::$ion_1_1::(
  (catalog_key "com.example.charts" 1)
  (load geo "com.example.geometry" 1)        ❶
  (macro_table
    (macro scatterplot
      [(points ':geo:point'...)]             ❷
      [points]))
)
```

❶ Loading the `geo` module means…

❷ …we can access `point` by qualified reference.

Here's another **`load`** clause, but this time it's inside a module rather than alongside them in an encoding directive. This makes the `geo` module visible only within this module, so we can reference `point` as the argument shape of the `scatterplot` macro. As before, we assign a symbolic name to the module for qualified references.

It's often preferable to avoid the clunky quoted qualified references by bringing into scope not just the `geo` module but also its macros, via **`use`**:

```
$ion_shared_module::$ion_1_1::(
  (catalog_key "com.example.charts" 1)
  (use (load geo "com.example.geometry" 1))      ❶
  (macro_table
    (macro scatterplot [(points point...)]        ❷
      [points]))
)
```

❶ Using the `geo` module means…

❷ …no qualification needed for `point`.

The **`use`** clause accepts a series of modules, by name or by **`load`**, and makes their exported macros visible in the body of the importing module. This is common, so there's a shorthand: `(import ...)` is equivalent to `(use (load ...))`.

Regardless of how `scatterplot` is declared, we know how to invoke it in a document:

```
$ion_1_1
$ion_encoding::(
  (load chart "com.example.charts" 1)
  (macro_table chart)
)
(:scatterplot (3 17) (395 23) (15 48) (2023 5))
```

While the signature of `point` is now implicit in the signature of `scatterplot`, and while the macro expander will invoke `point` while expanding `scatterplot`, neither `point` nor the module containing it is in scope within the document:

```
(:point 25 10)    ⇒ error: no installed module exports a macro named 'point'.
(:geo:point 2 1) ⇒ error: no module named 'geo' is installed.
```

In particular, `geo` is not in the encoding environment's available modules, since it wasn't imported into it:

```
$ion_1_1
$ion_encoding::(
  (load chart "com.example.charts" 1)
  (macro_table chart geo)
)
  ⇒ error: no module named 'geo' is available for installation.
```

When the Ion implementation loads the `chart` module, it will transitively load the geometry module as well, but the import of `com.example.geometry` by `com.example.charts` is *not visible by name* to the importer.

You can do similar things within an encoding directive:

```
$ion_1_1
$ion_encoding::(
  (module geo
    (macro_table
      (macro point [(x int), (y int)]
        {x: x, y: y})
      (macro line  [(a point), (b point)]
        [a, b])))
```

```
  (module chart
    (import geo)                                        ❶
    (macro_table
      (macro scatterplot [(points point...)]
        [points])))
  (macro_table chart)                                   ❷
)
```

❶ Importing `geo` makes its macros accessible within `chart`.

❷ The `geo` module is not installed into the encoding environment, so its macros are not accessible in the document body.

4.12 Macro Aliases

We've seen how to resolve an ambiguous macro name by using qualified references. Another approach is to give new names to existing macros. Suppose we want to add a 3d chart to our module, so we import both the 2d and 3d modules:

```
$ion_1_1
$ion_encoding::(
  (module chart
    (import geo "com.example.geometry" 1)
    (import g3d "com.example.graphics.3d" 1)
    (macro_table
      (macro scatterplot [(points point...)]

  ⇒ error: 'point' is ambiguous, exported by 'geo' and 'g3d'.
```

The most direct way to fix this is to use a qualified reference. We've seen this used in E-expressions like `(:geo:point 17 28)`, but now we need it in a signature where the special smile syntax does not apply. Instead, use a quoted symbol:

```
      (macro scatterplot [(points ':geo:point' ...)]
        [points]))
```

That has the intended effect of keeping `scatterplot` using 2D points, but it's somewhat awkward. A more ergonomic approach is to introduce an alias to disambiguate:

```
  (module chart
    (import geo "com.example.geometry" 1)
    (import g3d "com.example.graphics.3d" 1)
    (alias point2 ':geo:point')                       ❶
    (macro_table
      (macro scatterplot [(points point2 ...)]        ❷
        [points])
      ...
```

❶ Declaration of alias `point2`.

❷ Use of that new name in a signature.

Aliases can only be declared within a module, where they can be used wherever a macro reference occurs, including for macro invocations in the template language. In addition to disambiguation, they can be used to shorten long names, or to give names to anonymous macros.

4.13 Exports

Unlike `macro` definitions, aliases are not automatically exported from the module where they are declared; they are presumed to be implementation details. Sometimes it's helpful to make them available to consumers of the module, and for that they can be exported:

```
$ion_1_1
$ion_encoding::(
  (load geo "com.example.geometry" 1)
  (load g3d "com.example.graphics.3d" 1)
  (module local
    (alias point2 ':geo:point')   ❶
    (alias point3 ':g3d:point')
    (macro_table
      (export point2 point3)))
  (macro_table local geo g3d)
)
(:point2 93 5)
(:point3 0 12 33)
```

❶ Modules loaded at the directive level are visible within inline module bodies.

Exports can also be used to "pass through" selected macros from an imported module: `(export ':g2d:line')` exports the name `line` from the enclosing module. The pass-through form is *almost* the same as the pair of clauses:

```
(alias line ':g2d:line')
...
(export line)
```

...*except* the latter declares a local name while the pass-through does not.

> **Important**
> The macro names exported by a module must be unique, regardless of whether they are exported implicitly via **`macro`** or explicitly via **`export`**.

4.14 Extending the Macro Table

Some Ion use cases benefit from defining macros "on the fly" in response to repeated content. The techniques we used to extend the symbol table in Section 4.7 work for the macro table as well:

```
$ion_1_1
$ion_encoding::(
  (module mod1
    (symbol_table ["s1", "s2"])
    (macro_table (macro mac1 ...)))
  (symbol_table mod1)
  (macro_table  mod1)
)

// ... application data ...

$ion_encoding::(
  (retain *)
  (module mod2
    (symbol_table ["s3", "s4"])
```

```
    (macro_table (macro mac2 ...)))
  (symbol_table mod1 mod2)
  (macro_table  mod1 mod2)
)
```

4.15 Separate Installation

The preceding example has some repetition between **symbol_table** and **macro_table**, illustrating that the symbol and macro tables are maintained independently. The following is legal:

```
  (symbol_table mod1 mod2)
  (macro_table  mod2 mod1)
```

There's no assumption that the document needs both symbols and macros from every module, or that the relative allocation of addresses should be the same. If anything, we assume the opposite: that installing the macros from a module suggests that you don't need to install its symbols since they'll surface in the results of macro expansion.

If we find this particularly bothersome, a macro can eliminate the repetition:

```
(macro both_tables [(module_names symbol...)]
  (values
    (make_sexp (literal symbol_table) module_names)
    (make_sexp (literal macro_table ) module_names)))
```

Invoked as:

```
$ion_encoding::(
  (load foo ...)
  (load bar ...)
  (load baz ...)
  (:both_tables bar foo baz)
)
```

This leverages splicing to add two S-expressions to the enclosing directive.

4.16 Prioritization

The features we've explored can be combined to achieve fine-grained control over the allocation of macro and symbol addresses. This lets document authors assign the smallest opcodes to the most used macros and symbols.

Let's assume that our graphics modules have grown to include a large number of macros, far more than the 64 that can be invoked with a single-byte opcode. If we know that our document invokes, say, 3D `point` and `tri` more than anything else, we can grant them single-byte opcodes by ensuring they show up first among the installed macros:

```
$ion_1_1
$ion_encoding::(
  (load geo "com.example.geometry" 1)
  (load g3d "com.example.graphics.3d" 1)
  (module priority
    (use g3d)
    (macro_table
      (export point tri)))
  (macro_table priority geo g3d)
)
(:0 101 17 5)                           // invoke :g3d:point
(:1 (101 17 5) (101 17 20) (100 17 20))  // invoke :g3d:tri
```


Chapter 5

Encoding Directives

TODO
This intro section is probably misplaced in the context of the larger book. Move or integrate elsewhere.

Ion 1.0 uses symbol tables to capture and compress repeated symbol text. At all points in an Ion document, there exists an *encoding environment* that contains the *current symbol table* mapping symbol IDs to text. The encoding environment of a document is controlled by directives embedded in the document. These directives are encoding artifacts and not part of the application data model. Ion 1.0 has two directive forms:

- An Ion Version Marker (IVM) resets the environment to the default provided by that version of Ion.
- An `$ion_symbol_table` struct defines a new environment that takes effect immediately after the struct closes.

The latter form includes a feature that allows the new environment to be specified in terms of the current one in a limited fashion: the current symbol table can be imported as if it were shared, so that new symbols can be appended to it.

To increase compression across many documents that have similar content (for example, they use the same schema), Ion 1.0 has *shared symbol tables* that capture a portion of an encoding context—a list of symbols—that can be imported into many local symbol tables.

Ion 1.1 generalizes and refactors these features:

- *Macros* are a generalization of symbols in the sense that they are a feature to enable increased density. The Ion parser expands integer symbol IDs to symbol text; it now also expands macro expressions into data of arbitrary type and cardinality.
- The encoding context is extended to contain a *local macro table* alongside the local symbol table. In much the same way that the local symbol table defines an address-space for identifying symbols, the local macro table defines an address-space for macros.
- Symbols and macros are defined and collected inside *encoding modules*. Modules subsume shared symbol tables while remaining backwards-compatible with their current data model and catalog semantics. Any existing Ion shared symbol table is a valid encoding module, albeit one that only declares symbols and not macros.
- The `$ion_symbol_table` struct and its behavior are subsumed by a new `$ion_encoding` top-level S-expression that imports and defines modules, then separately assembles the local symbol and macro tables.

This chapter focuses on the new components of the top-level context and the `$ion_encoding` S-expression that controls it.

5.1 Document Structure

TODO Cover document segmentation, environment components, etc.

TODO The below should probably move elsewhere

As we've seen, encoding directives manipulate the global context, managing modules and installing (some of) them into the local symbol and macro tables. To clarify this behavior, we should first discuss the lifecycle of modules. An Ion 1.1 implementation must manage a few distinct sets of modules:

- The *loaded modules* are those that have been defined or loaded by an encoding directive, or transitively loaded from another loaded module.
- The *available modules* are loaded modules that have been assigned a name via a `load` or `import` clause.
- The *installed modules* are available modules that have been listed in an encoding directive's `symbol_table` or `macro_table` field. Technically, modules are installed immediately following termination of the `$ion_encoding` directive.

Each encoding directive on the stream fully replaces the prior context. A user module becomes unavailable when a succeeding directive fails to retain it explicitly. A loaded module can be unloaded (garbage collected) when its no longer reachable from an available module.

5.2 Ion Version Markers

The bootstrap directive, required at the start of all Ion 1.1 segments, and acceptable mid-stream, is the Ion version marker:

```
$ion_1_1
```

This keyword has the effect of resetting the encoding context to the default modules, symbols, and macros provided by the Ion specification. More precisely, the default context has a single available module named `$ion`, installed for both symbols and macros. This ensures that the system symbols and system macros provided by Ion 1.1 are available by default.

The system module and its macros are in fact available everywhere in the document, and cannot be removed or redefined by `$ion_encoding`: to a large degree, it's as if the **`retain`**, **`symbol_table`**, and **`macro_table`** clauses all have `$ion` as their implicit first element. As a result, system macros can always be invoked by `(:$ion:`*`name`*` ...)`.

System macros have one additional bit of special handling: they are binary-encoded using a dedicated opcode, using a dedicated address space that's independent of the explicitly-enumerated modules in **`macro_table`**. This means that the initial range of unqualified numeric macro references like `(:3 ...)` don't inherently refer to system macros. User-level macros get priority to those precious single-byte opcodes.

5.3 `$ion_encoding` Directives

The `$ion_encoding` directive declares a set of available modules, then assembles some subset of those into the local symbol and macro tables. The general shape of an encoding directive is as follows:

```
$ion_encoding::(
  (retain ...)        // Reuse selected modules from the current segment
  (load ...)          // Get a shared module from the catalog
  (module ...)        // Define a new module inline
  ...
  (symbol_table ...)  // Install modules into the symbol table
  (macro_table ...)   // Install modules into the macro table
)
```


More formally, here's the relevant portion of the domain grammar:

encoding-directive	::=	`$ion_encoding::(` *retention*? *module-decl** *symtab*? *top-mactab*? `)`

The directive has four sections: declare currently available modules to retain, declare additional modules to make available, define the new symbol table, define the new macro table.

5.3.1 Retaining Available Modules

An encoding directive defines a new encoding environment in terms of the current environment (that is, the encoding environment for the segment containing the directive). By default, the new environment starts with an empty set of available modules, and if any modules are to be reused by the new segment, they must be explicitly retained.

retention	::=	`(` **`retain`** *retainees* `)`
retainees	::=	`'*'` | *module-name**

Before declaring new names, the directive can selectively **`retain`** available modules (that is, modules declared in the preceding directive. This is done either by using the keyword **`*`** to copy all available modules from the current encoding environment into the new one, or by enumerating specific names to copy.

5.3.2 Declaring Modules

After possibly retaining modules from the current environment, the directive can make additional modules available, either loading them from the implementation's catalog, or defining them inline. Either way, an entry is added (or updated) in the directive's map of available modules.

module-decl	::=	*dependency* | *inline-module-def*
dependency	::=	*load-decl* | *use-decl* | *import-decl*

The names of available modules can be remapped: if a name is reused, the earlier declaration is shadowed through the rest of the directive (including upcoming inline modules).

5.3.2.1 Loading Shared Modules

To make a shared module available, it must first be *loaded*, which gives the module a symbolic name that can be used to reference the module's components.

load-decl	::=	`(` **`load`** *load-body* `)`
load-body	::=	*module-name catalog-name catalog-version symbol-maxid*?
catalog-name	::=	*unannotated-string*
catalog-version	::=	*unannotated-uint*
symbol-maxid	::=	*unannotated-uint*

This works like an `import` struct in `$ion_symbol_table` in that it acquires an entity from the implementation's catalog, though here there is no direct effect on the symbol table. The *catalog-name*, *catalog-version*, and *symbol-maxid* arguments have the same meaning as the corresponding fields of an `imports` struct, but only the latter is optional. Resolving the name and version to a shared module is the same as for shared symbol tables, using the same algorithm for inexact match on the version.

Tip

A primary design tenet of Ion 1.1 is to remain compatible with existing catalog APIs and services that vend shared symbol tables. Existing shared symbol tables are shared modules that export no macros.

As suggested by its name, the *symbol-maxid* argument only affects symbol allocation, not macros. Use of macros from a shared module requires exact match of the *shared-version*, and a module that was imported inexactly will trigger an error if its name appears within a local module or **`macro_table`**.

5.3.2.2 Defining Inline Modules

Along with loading shared modules, a directive can define local modules. From the perspective of the rest of the encoding directive, and the data that follows, there's no meaningful distinction in the result. Either way, there's another module available for use.

inline-module-def	::=	(**module** *module-name module-body*)

TODO import and link to the module reference.

Note that module names are lexically scoped: an inline module's body can access modules previously made available by the enclosing directive. That is, their macros can be accessed by qualified references, but unqualified references require a **`use`** clause in the module of directive.

5.3.3 Using Modules

In the context of an encoding directive, a **`use`** clause makes macros visible within upcoming inline modules, so they can be referenced without qualification (assuming no ambiguity).

use-decl	::=	(**use** *use-item**)
use-item	::=	*module-name* | *load-decl*

You can use a module by name, referring to a previously retained, loaded, or inline module, or in combination with **`load`**. In the latter case, `(use (load module ...))` is equivalent to `(load module ...)(use module)`.

TODO This is incorrect: An Ion parser must signal a fatal error if a directive uses a shared module that cannot be acquired by exact match to the declared catalog version.

The **`import`** clause is simply a shorthand for "load and use".

import-decl	::=	(**import** *load-body*)

That is, `(import module ...)` is equivalent to `(load module ...)(use module)`.

5.3.4 Assembling the Symbol Table

Modules must be installed into the symbol table to affect the encoding of symbols.

symtab	::=	(**symbol_table** *symtab-item**)
symtab-item	::=	*module-name* | [*text**]

TODO update this

The **`symbol_table`** field is simply a list of module names, with no duplicates allowed. The `$ion` module is implicitly first in the list and cannot be named explicitly. All names must be in the declared earlier in the directive, including implicit inclusion via `(retain *)`

The effect of this field is to allocate addresses to symbol text, in a manner identical to Ion 1.0 imports, allocating contiguous ranges to each installed module. The width of each range is the number of symbols exported by the corresponding module, or the *symbol-maxid* argument of the associated **`load`** clause, when provided.

In encoded data numeric symbol references (in text, using the form `$d+`) work the same way as in Ion 1.0: the first system symbol is `$1` and the first user-installed module starts where the system symbols end.

5.3.5 Assembling the Macro Table

TODO This needs work.

Modules must be installed into the macro table to enable their use in the document's E-expressions.

top-mactab ::= (**macro_table** *module-name**)

The meaning is nearly identical to that of **symbol_table** in that it allocates macro addresses by effectively concatenating the exported macro tables of the listed modules.

The differences versus symbols are:

- The names of modules installed for macros are part of the "macro environment" of the new encoding context, and are used to resolve qualified macro references. The names of modules in **symbol_table** are not added to the context's visible environment and cannot be used to reference symbols.
- Shared modules that are in **macro_table** must have exactly the version requested.
- There's no corollary to *symbol-maxid* for macro imports.

5.4 $ion_symbol_table Directives

TODO This content is very old and needs much attention.

Ion 1.1 still supports the legacy `$ion_symbol_table` directive, internally transforming it into an equivalent `$ion_encoding` form.

This is generally not detectable by users, except when followed by an `$ion_encoding` directive that retains the current modules. In that case, the imported symbol tables, and the synthetic local module, are visible to the new encoding context, so we must TODO define what those names are.

TODO define the transformation formally

Chapter 6

Encoding Modules

6.1 Overview

6.1.1 Module Interface

The interface to a module consists of:

- its *spec version*, denoting the Ion version used to define the module
- its *exported symbols*, an array of strings denoting symbol content
- its *exported macros*, an array of <name, macro> pairs, where all names are unique identifiers (or null).

The spec version for an inline module is implicitly derived from the Ion version of its containing segment. The spec version for a shared module is denoted via a required annotation.

The exported symbol array is denoted by the **`symbol_table`** clause of a module definition, and by the `symbols` field of a shared symbol table.

The exported macro array is denoted by the module's **`macro_table`** clause, with addresses allocated to bindings in the order they are declared. One address is allocated per **`macro`** definition, while the **`export`** clause allocates one address for each listed macro.

6.1.2 Internal Environment

The body of a module tracks an internal environment by which macro references are resolved. This environment is constructed incrementally by each clause in the definition and consists of:

- the *visible modules*, a map from identifier to module
- the *imported macros*, a map from identifier to macro (or to an ambiguity sentinel)
- the *local macros*, a map from identifier to a macro and optional exported address.
- the *exported macros*, an array containing name/macro pairs

Before any clauses of the module definition are examined, the initial environment is as follows:

- The visible modules map `$ion` to the system module for the appropriate spec version. For an inline module, it also includes the modules previously made available by the enclosing encoding directive (via **`retain`**, **`load`**, or **`import`**).

- The imported macros contain the exported macros from that system module. For an inline module, it also contains the exported macros from modules previously **use**d or **import**ed by the enclosing encoding directive.
- The exported macros and local macros are empty.

The first section of a module definition consists of dependency declarations in the form of **use**, **load**, and **import** clauses. This section affects the environment as follows:

- A **load** declaration retrieves a shared module from the implementation's catalog and assigns it a name in the visible modules. An error must be signaled if the name already appears in the visible modules.
- A **use** declaration adds its arguments to the visible modules, and adds their exported macros to the imported macros. When a name is exported from more than one module, and refers to different macros, its mapping points to a sentinel value recording the ambiguity.
- An **import** declaration is shorthand for loading a shared module and immediately using it.

After these dependencies are declared, a **symbol_table** definition may follow.

Next, any number of **alias** declarations.

- An **alias** clause associates a (presumably) new name with an existing macro. An error must be signaled if the name exists in the local macros. Otherwise, the name is added to the local macros.

Finally, there's the **macro_table** definition, affecting the local macros and the exported macros.

- An **export** clause exports imported and aliased macros. Each entry in the clause is handled in order. If the given reference is anonymous, the macro is appended to the exported macro array without a name. When the reference uses a name, an error must be signaled if it already appears in the exported macro array. Otherwise, the name and macro are appended to the exported macro array.
- A **macro** clause defines a new, exported macro. An error must be signaled if the definition uses a name that exists in the local macros. Otherwise, the name and macro are appended to the exported macro array, and (when not anonymous) the name, macro, and address are added to the local macros.
- A module name TODO

6.2 Resolving Macro References

Within a module definition, macros can be referenced in several contexts using the following *macro-ref* syntax:

macro-ref	::=	*macro-name* | *local-ref* | *qualified-ref*
local-ref	::=	⟨symbol of the form `':`*name-or-address*`'` ⟩
qualified-ref	::=	⟨symbol of the form `':`*module-name*`:`*name-or-address*`'` ⟩
module-name	::=	*unannotated-identifier-symbol*
macro-name	::=	*unannotated-identifier-symbol*
macro-address	::=	*unannotated-uint*
name-or-address	::=	*macro-name* | *macro-address*

Macro references are resolved to a specific macro as follows:

- An unqualified *macro-name* is looked up within the local macros, and if not found then the imported macros. If it maps to a macro, that's the resolution of the reference. Otherwise, if the name maps to the ambiguity sentinel, an error is signaled due to an ambiguous reference. Otherwise, an error is signaled due to an unbound reference.
- A named local reference (`':name'`) is looked up within the local macros. If there's no entry, an error is signaled due to an unbound reference.

- An anonymous local reference (`':address'`) is resolved by index in the exported macro array. If the address exceeds the array boundary, an error is signaled due to an invalid reference.
- A qualified reference (`':module:name-or-address'`) resolves solely against the referenced module. If the module name does not exist in the visible modules, an error is signaled due to an unbound reference. Otherwise, the name or address is resolved within that module's exported macro array.

Note
An unqualified macro name can change meaning in the middle of a module: it could be imported and used with that meaning, then a declaration shadows that name and gives it a new meaning.

6.3 Module Versioning

Every module definition has a *spec version* that gives the definition its meaning in terms of acceptable syntax, available features, and so on. A module's spec version is expressed in terms of a specific Ion version; the meaning of the module is as defined by that version of the Ion specification.

The spec version of a shared or tunneled module must be declared explicitly using an annotation of the form `$ion_1_N`. This allows the module to be serialized using any version of Ion, and its meaning will not change.

```
$ion_shared_module::$ion_1_1::(
  (catalog_key "com.example.symtab" 3)
  (symbol_table ...)
  (macro_table ...)
)

$ion_shared_symbol_table::{
  name: "com.example.symtab", version: 3,
  symbols: [...],
  module: $ion_1_1::(      // Spec version is 1.1
    // Semantics of this module are specified by Ion 1.1, regardless of the
    // enclosing document's Ion version.
    ...
  )
}
```

The spec version of an inline module is always the same as the Ion version of its enclosing segment.

```
$ion_1_1
$ion_encoding::(
  (module M1 ...)     // Module semantics specified by Ion 1.1
  ...
}
...
$ion_1_3
$ion_encoding::(
  (module M2 ...)     // Module semantics specified by Ion 1.3
  ...
}
...                   // Assuming no IVM
$ion_encoding::(
  (module M3 ...)     // Module semantics specified by Ion 1.3
  ...
}
```

To ensure that all consumers of a module can properly understand it, a module can only import shared modules defined with the same or earlier spec version.

6.4 Inline, Shared, and Tunneled Modules

Inline modules are defined within an `$ion_encoding` directive, and are available only within the enclosing document. Their scope is lexical; they can be used immediately following their definition, up until the next directive, at which point they'll either be retained by the new encoding environment, or made unavailable.

inline-module-def	::=	(**`module`** *module-name module-body*)

Inline modules always have a symbolic name given at the point of definition. They inherit their spec version from the surrounding document, and they have no content version.

Shared modules exist independently of the documents that use them. They are identified by a *catalog key* consisting of a string name and an integer version. When consumed by a document or another module, they are given a local identifier.

shared-module-def	::=	`$ion_shared_module::`*ion-version-marker*`::` (*catalog-key module-body*)
catalog-key	::=	(**`catalog_key`** *catalog-name catalog-version*)

Tunneled modules are shared modules that are defined within a shared symbol table definition.

shared-symtab	::=	`$ion_shared_symbol_table::{ name :` *catalog-name* `version :` *catalog-version* `symbols : [` string* `] module :` *tunneled-module-def* `}`
tunneled-module-def	::=	*ion-version-marker* `::` (*tunneled-module-body*)
tunneled-module-body	::=	*dependency* macro-alias* module-mactab*

Shared and tunneled modules have self-declared catalog-names that are generally long, since they must be more-or-less globally unique. That's not usable as a namespace qualifier, so they are given local symbolic names by load and import declarations. They have a spec version that's explicit via annotation, and a content version derived from the catalog version.

6.5 Module Bodies

The body of a module is a sequence of elements following this grammar:

module-body	::=	*dependency* symtab? macro-alias* module-mactab?*

6.5.1 Dependencies

Inline modules automatically have access to modules previously declared in the enclosing directive using `retain`, `module`, `load`, or `import`. Macro names are also visible as declared by directive-level `use` and `import` clauses. Shared and tunneled modules lie outside an encoding directive and have no such automatic visibility into other modules.

To extend any such automatic names within a module body, you can write the same `load`, `use`, and `import` clauses that are acceptable within an `$ion_encoding` directive. The difference is one of scope: the module and macro names introduced by these forms only affect the enclosing module, not the overall encoding environment.

dependency	::=	*load-decl* | *use-decl* | *import-decl*

6.5.2 The Symbol Table

A module can define a list of exported symbols by copying symbols from other modules and/or declaring new symbols.

symtab	::=	(**symbol_table** *symtab-item**)
symtab-item	::=	*module-name* | [*text**]

Note
This clause is not allowed in tunneled modules.

This clause builds a list of symbol-texts by concatenating the elements (the symbol tables of named modules, and the lists of symbol/string values).

Where a module name occurs, that module must have been previously loaded in the enclosing module or encoding directive, and its symbol table is appended. If a *symbol-maxid* was given when loaded, the list is truncated or padded to that length.

Where a list occurs, it follows the syntax and semantics to the symbols field of `$ion_shared_symbol_table`. In addition, it allows symbols as well as strings.

TODO: "inline" the specified behavior of such lists.

6.5.3 Declaring Macros

Macros are declared after symbols, in two parts. First, a set of aliases, then the macro table itself.

A macro name is a symbol that can be used to reference a macro, both inside and (if public or exported) outside the module. Macro names are optional, and improve legibility when using, writing, and debugging macros.

When a name is used, it must be an identifier per Ion's syntax for symbols. If the name is also exported by any visible module, the import is shadowed by the declaration. An error must be signaled if the same macro name occurs more than once among the declarations.

TODO: the above repeats content from elsewhere.

6.5.3.1 Macro Aliases

Aliases simply create a new name bound to an existing macro.

macro-alias	::=	(**alias** *macro-name macro-ref*)

```
(alias s some_long_name)
(alias t ':some_module:23')              // Give name to an anonymous macro
```

The effect of an alias is to resolve the reference to determine the corresponding macro, and to assign a name for it in the local macro map.

Unlike **`macro`** definitions, aliases are not implicitly exported, do not have addresses allocated, and cannot be referenced using **:***`address`* syntax. If an alias is later exported, an address is allocated at that time.

6.5.3.2 Macro Definitions

After aliases, a macro table can be defined.

module-mactab	::=	(**macro_table** *macro-or-export**)
macro-or-export	::=	*macro-defn* | *export*

Most commonly, a macro table entry is a definition of a new macro expansion function, following this general shape:

macro-defn ::= (**macro** *macro-name*? *signature template*)

When no name is given, this defines an anonymous macro that can be referenced by its numeric address (that is, its index in the enclosing macro table). Inside the defining module, that uses a local reference like `':12'`.

The *signature* defines the syntactic shape of expressions invoking the macro; see TODO for details. The *template* defines the expansion of the macro, in terms of the signature's parameters; see Chapter 9 for details.

6.5.3.3 Exporting Macros

Aliases and **use**d or **import**ed macros and aliases must be explicitly exported if so desired. Export clauses can be intermingled with **macro** definitions inside the **macro_table**; together, they determine the bindings that make up the module's exported macro array.

Exports are expressed in two ways: **export** clauses and module names:

export ::= (**export** *export-item**) | *module-name*
export-item ::= *macro-ref* | (**from** *module-name name-or-address**)

An **export** clause contains a sequence of macro references, using the normal single-symbol syntax, or an S-expression variant that exports multiple macros from the same module. Each entry in the clause is handled in order.

Where a macro-ref appears, the referenced macro is appended to the macro table. When the reference uses an address, the macro is exported without a name. When the reference uses a name, an error must be signaled if it already appears in the macro table.

A **from** clause is shorthand for a series of qualified references from within a single module.

The *module-name* export form is shorthand for referencing all exported macro from that module, in their original order.

Tip
No name can be repeated among the exported macros, including macro definitions. Name conflicts must be resolved by aliases.

Chapter 7

Macro Signatures

A macro's *signature* defines the syntax of expressions that invoke it, and the set of input values it accepts. Signatures apply to both E-expressions and macro-language invocations. Because they denote the interface for users of macros, we describe them independently of macro definitions.

A signature consists of a sequence of named parameter specifications, followed by an option result specification.

signature	::=	*param-specs result-spec*?
param-specs	::=	(*param-spec* rest-spec*?) | [*param-spec* rest-spec*?]
param-spec	::=	*param-name* | (*param-name param-shape*)
rest-spec	::=	(*param-name rest-shape*)
param-name	::=	*unannotated-identifier-symbol*

Each parameter in a signature has a name, expressed as a Ion identifier symbol. Restricting names to Java-style identifiers enables use of operator characters (like `?` and `*`) for the syntax surrounding names, including qualified macro references.

7.1 Parameter Shapes

A macro's "wire format"—the sequence of acceptable tokens in an E-expression—is determined by its parameters' shapes. The shape of a parameter has two dimensions: its *base type* and its grouping. The base type constrains the expression forms that can be used for each argument supplied to the parameter. Independently, a parameter is either *simple*, *grouped*, or a *rest parameter*; this dimension determines how the arguments supplied to the parameter are delimited within the overall invocation.

param-shape	::=	*simple-shape* | *grouped-shape*
simple-shape	::=	*tagged-type*? *tagged-cardinality*? | *tagless-type tagless-cardinality*?
grouped-shape	::=	[*any-type*?] *grouped-cardinality*?
rest-shape	::=	*any-type*? *rest-cardinality*

7.2 Base Types

The core of a parameter specification is its base type, which constrains the syntax of each argument (that is, the acceptable expression forms that can be used).

any-type	::=	*tagged-type* | *tagless-type*
tagged-type	::=	*abstract-type* | *concrete-type*
tagless-type	::=	*primitive-type* | *macro-ref*

The *concrete types* correspond to the usual Ion data types, from `null` and `bool` through `list` and `struct`. These have the obvious meanings, with the caveat that annotations are allowed, as are appropriately-typed *and untyped* nulls. For example, the inputs `null.int` and `null.null` are acceptable to an `int`-typed argument, as are arbitrary annotations on either.

concrete-type	::=	**'null'** | **bool** | **timestamp** | **int** | **decimal** | **float** | **string** | **symbol** | **blob** | **clob** | **list** | **sexp** | **struct**

The *abstract types* are select supertypes of the concrete types: `text` accepts both `symbol` and `string`; `number` accepts `int`, `decimal`, and `float`; `lob` accepts `blob` and `clob`; `sequence` accepts `list` and `sexp`; `any` accepts any value. Nulls and annotations are accepted as with the concrete types.

abstract-type	::=	**any** | **number** | **exact** | **text** | **lob** | **sequence**

Collectively, the abstract and concrete types are the *tagged types*. Parameters of these types can use macro invocations in place of normal values.

The *primitive types* are subtypes of various concrete types that have particularly compact binary encodings. These include variable-length strings, symbols, signed `ints`, unsigned `uints`, as well as fixed-width `ints`, `uints`, and `floats`, all of various widths between 8 and 64 bits. These types are untagged, so they do not accept nulls, annotations, or macro invocations.

primitive-type	::=	**var_symbol** | **var_string** | **var_int** | **var_uint** | **uint8** | **uint16** | **uint32** | **uint64** | **int8** | **int16** | **int32** | **int64** | **float16** | **float32** | **float64**

Finally, any visible macro can be used as a type, in which case the argument is written (in text) as an S-expression with elements matching that macro's signature. As with tagless arguments, these arguments are serialized without any explicit indication of their type, since that's implied by context. (Using a zero-parameter macro as a parameter type is acceptable but pointless, since the result is constant.)

7.3 Cardinality

Each parameter specification includes a *cardinality* that indicates the number of values that it expects its argument(s) to produce.

tagged-cardinality	::=	**!** | **+** | **'?'** | **'*'**
tagless-cardinality	::=	**'?'**
grouped-cardinality	::=	**'+'**
rest-cardinality	::=	**...** | **...+**

The following cardinality modifiers are available:

- `?` denotes a parameter that accepts zero or one value.
- `!` denotes a parameter that accepts exactly one value.
- `*` denotes a parameter that accepts zero or more values.
- `+` denotes a parameter that accepts one or more values.

Cardinality is verified by the Ion implementation: the expansion system will signal an expansion error if the number of values produced by the argument(s) is not aligned with the declared cardinality.

Some combinations of type and cardinality are inherently erroneous: a primitive type cannot produce more than one value.

7.4 Grouped Parameters

A parameter may be *grouped*, in which case its invocation shape is a sequence of arguments. In text invocations, this sequence is written as an Ion list containing the arguments. In binary E-expressions, the sequence uses a dedicated encoding. In all cases, each element of the group must match the parameter's declared type.

TODO expansion splicing semantics

To declare a grouped parameter, write the parameter specification with a list around the base type. Grouped parameters may declare the + cardinality, otherwise `*` is implied. No other cardinalities are allowed; there's no point in grouping a parameter that accepts at most one value.

Examples:

```
(counts [int])          // Accepts zero or more ints
(points [point]+)       // Accepts one or more points
```

7.5 Rest Parameters

The last parameter may be a *rest parameter*, which is effectively an implicitly grouped parameter. In text invocations, these parameters don't use a grouping sequence, but instead take "all the rest" of the argument expressions.

To declare a rest parameter, use one of the two special cardinality modifiers:

- `...` denotes a parameter that accepts zero or more values.
- `...+` denotes a parameter that accepts one or more values.

Examples:

```
(counts int ...)      // Accepts zero or more ints
(points point ...+)   // Accepts one or more points
```

7.6 Voidable and Optional Parameters

Parameters with cardinality accepting zero values (declared with modifiers `?`, `*`, or `...`) are called *voidable* because their resulting value streams can be void. A parameter is *optional* when it is voidable and all following parameters are voidable.

Optional parameters are given special treatment in text invocations: their arguments can be omitted entirely (as long as all following arguments are also omitted).

7.7 Arity

The *minimum arity* of a macro is equal to the number of leading non-optional parameters. Assuming no rest-parameter, the *maximum arity* of the macro is the total number of declared parameters. A macro with a rest-parameter has no maximum arity. A macro with equal minimum and maximum arity is *fixed arity*; other templates are *variable arity*.

7.8 Result Specification

To enable more robust and easier-to-debug templates, a signature can express a *result specification* that constrains the data that it produces. Results are specified by their type (abstract or concrete) and cardinality. Both factors are verified by the macro expander when the macro is invoked.

result-spec	::=	**->** *tagged-type tagged-cardinality*

Chapter 8

The System Module

The symbols and macros of the system module `$ion` are available everywhere within an Ion document, with the version of that module being determined by the spec-version of each segment.

The specific system symbols are largely uninteresting to users; while the binary encoding heavily leverages the system symbol table, the text encoding that users typically interact with does not. The system macros are more visible, especially to authors of macros.

This chapter catalogs the system-provided macros. The examples below use unqualified names, which works assuming no other module exports the same name, but the unambiguous form `:$ion:`*`macro-name`* is always correct.

Important
This list is not complete. We expect it to grow and evolve as we gain experience writing macros.

8.1 Primitive Operators

This section describes operators that cannot be defined as macros.

8.1.1 Stream Constructors

8.1.1.1 `void`

```
(void) -> any?
```

Produces an empty stream. The most common use of this operator is to supply “no value” to a voidable parameter. To make such use more readable, the special-case E-expression `(:)` is synonymous to `(:void)`.

8.1.1.2 `values`

```
(values (v any...)) -> any*
```

Produces a stream from any number of arguments, concatenating the streams produced by the nested expressions. Used to aggregate multiple values or sub-streams to pass to a single argument, or to return multiple results. Generally only useful with more than one subexpression.

8.1.2 Value Constructors

8.1.2.1 make_string

```
(make_string (content text...)) -> string
```

Produces a non-null, unannotated string containing the concatenated content produced by the arguments. Nulls and annotations are discarded.

TODO https://github.com/amazon-ion/ion-docs/issues/255 Probably useful to allow some other Ion scalars (at least) to allow type conversion. I think this would be most useful for ints, since the binary representation is more compact than as characters. Lobs wouldn't work well, though.

8.1.2.2 make_symbol

```
(make_symbol (content text...)) -> symbol
```

Like `make_string` but produces a symbol.

8.1.2.3 make_list

```
(make_list (vals any...)) -> list
```

Produces a non-null, unannotated list from any number of inputs. Template expressions of the form `[`E_1`, ..., `E_n`]` are equivalent to `(make_list `E_1` ... `E_n`)`.

8.1.2.4 make_sexp

```
(make_sexp (vals any...)) -> sexp
```

Like `make_list` but produces a sexp. This is the only way to produce an S-expression from a template: unlike lists, S-expressions in templates are not quasi-literals.

```
(:make_sexp)      ⇒ ()
(:make_sexp null) ⇒ (null)
```

8.1.2.5 make_struct

```
(make_struct (kv any...)) -> struct
```

Produces a non-null, unannotated struct from any number of elements. The `kv`s are processed in order, incrementally adding fields to an initially-empty struct. Various forms of `kv`s are allowed:

- A (non-null) string or symbol is treated as a field name, and MUST be followed by another value to comprise a key-value pair in the result. Annotations on the field name are discarded.
- A (non-null) struct is merged into the result as-is, after discarding annotations.
- Any other type of value evokes an expansion error.

Template expressions of the form `{`T_1`:`E_1`, ..., `T_n`:`E_n`}` are equivalent to `(make_struct (`**`literal`**` `T_1`) `E_1` ... (`**`literal`**` `T_n`) `E_n`)`, assuming that no expression E produces more than one value. In that case, the `make_struct` variant would misbehave: the second value produced by E would be treated as the next key.


```
(:make_struct k1 1 k2 2 {k3:3} k4 4)  ⇒  {k1:1, k2:2, k3:3, k4:4}
```

Because rest-parameters receive the concatenated argument result-streams, `make_struct`'s key-value pairs may not align with the actual arguments. This is different from splicing of macro results into structs, causing the key to repeat:

```
{ k1: (:values 1 k2) }                ⇒  { k1: 1, k1: k2 }
(:make_struct k1 (:values 1 k2) 2)   ⇒  { k1: 1, k2: 2 }
```

8.1.2.6 make_decimal

```
(make_decimal (coefficient int) (exponent int)) -> decimal
```

Since decimal is already compact, this is perhaps most useful in conjunction with packed arrays, or when the exponent is repeated and can be baked into a macro.

TODO https://github.com/amazon-ion/ion-docs/issues/253 If the coefficient were decimal, this could re-scale values. Useful?

8.1.2.7 make_float

```
(make_float ieee) -> float
```

Included for completeness, but of unclear utility.

TODO https://github.com/amazon-ion/ion-docs/issues/252 Coerce an int or decimal to float? Perhaps useful to use fixed-width ints to encode various float widths? This may not be useable to convert "IEEE bits" to float, since they would be converted to int before arriving here.

8.1.2.8 make_timestamp

```
(make_timestamp
  (year int) (month? int) (day int?)
  (hour int?)  (minute int?) (second decimal?)
  (offset int?))
  → timestamp
```

Produces a non-null, unannotated timestamp at various levels of precision. When `offset` is absent, the result has unknown local offset; offset `0` denotes UTC.

TODO https://github.com/amazon-ion/ion-docs/issues/256 Reconsider offset semantics, perhaps default should be UTC.

Example:

```
(macro ts_today
  ((hour uint8) (minute uint8) (seconds_millis uint32))
  (make_timestamp 2022 04 28 hour minute
    (decimal seconds_millis -3) 0))
```

8.1.2.9 annotate

```
(annotate (ann [text]*) value) -> any
```

Produces the `value` prefixed with the annotations `anns`. Each `ann` must be a non-null, unannotated string or symbol.

```
(:annotate ["a2"] a1::true) ⇒ a2::a1::true
```


8.2 Derived Operators

These operators can be defined in terms of the primitives, using the macro language.

8.2.1 Symbol Table Management

8.2.1.1 Local Symtab Declaration

This macro is optimized for representing symbols-list with minimal space.

```
(macro import
  ((name string) (version uint?) (max_id uint?)) -> struct
  { name:name, version:version, max_id:max_id })

(macro local_symtab
  ((imports [import]) (symbols string...))
  $ion_symbol_table::{
    imports:(if_void imports (void) [imports]),
    symbols:(if_void symbols (void) [symbols]),
  })
```

```
(:local_symtab [("my.symtab" 4)] "newsym" "another")
  ⇒
  $ion_symbol_table::{ imports:[{name:"my.symtab", version:4}],
                       symbols:["newsym", "another"] }
```

8.2.1.2 Local Symtab Appending

```
(macro lst_append
  ((symbols string...))
  (if_void symbols
    (void)                  // Produce nothing if no symbols provided.
    $ion_symbol_table::{
      imports: (literal $ion_symbol_table),
      symbols: [symbols]}))
```

```
(:lst_append "newsym" "another")
  ⇒
  $ion_symbol_table::{ imports:$ion_symbol_table,
                       symbols:["newsym", "another"] }
```

8.2.1.3 Embedded Documents (aka Local Scopes)

TODO

8.2.2 Compact Module Definitions

TODO

Chapter 9

Template Expressions

The behavior of a macro is defined in terms of an expression language. Like encoding directives and modules, this language is expressed as Ion data, and the meaning of templates is defined structurally and recursively based on the Ion data model.

9.1 Grammar

Here's the relevant portion of the domain grammar:

template	::=	*identifier* | *literal* | *quasi-literal* | *special-form* | *macro-invocation*
literal	::=	*null* | *bool* | *int* | *float* | *decimal* | *timestamp* | *string* | *blob* | *clob*
quasi-literal	::=	`[` *template** `]` | `{` *quasi-field** `}`
quasi-field	::=	*text* `:` *template*
special-form	::=	`(` **`literal`** *datum* `)` | `(` **`if_void`** *template*$_{cond}$ *template*$_{then}$ *template*$_{else}$ `)` | `(` **`if_single`** *template*$_{cond}$ *template*$_{then}$ *template*$_{else}$ `)` | `(` **`if_many`** *template*$_{cond}$ *template*$_{then}$ *template*$_{else}$ `)` | `(` **`for`** `[` *for-clause** `]` *template*$_{body}$ `)`
for-clause	::=	`(` *identifier template*$_{in}$ `)`
macro-invocation	::=	`(` *macro-ref macro-arg** `)`
macro-arg	::=	*template* | `[` *template** `]` // *Very* roughly

An expression in this language is called a *template*, and the expansion of a template (that is, its evaluation) produces a stream of Ion values. The central design concept is that symbols denote variable references, S-expressions denote operator invocations, and other Ion types denote values of that type.

9.1.1 Symbols are Variable References

When a template is an Ion symbol, it denotes a reference to a variable, either a macro parameter or a local binding from a `for` expression. The result of this template is the stream of values referred to by that variable.

The symbols used for variable names must be *identifiers* as defined by the Ion specification: a sequence of ASCII letters, digits, or the characters `$` (dollar sign) or _ (underscore), not starting with a digit.

When a template is expected, the symbols `$0` and `null.symbol` evoke a syntax error, as does any annotated symbol.

> **Tip**
> To denote the literal symbol `foo`, use the template `(`**`literal`**` foo)`.

9.1.2 Other Scalars are Literals

When a template is a non-symbol Ion scalar, it denotes a literal value, and the template expands into that value. Any annotations on the template are included in the output.

9.1.3 Lists and Structs are Quasi-Literals

When a template is an Ion list or struct, it denotes a quasi-literal of the same type. We say “quasi” literal because the elements of the container are treated as templates, not literal values.

When a template is a list, it expands into a list with the same annotations. The elements of the list-template are each treated as templates themselves. Each sub-template may produce any number of values, and the resulting streams are all concatenated to produce the output list.

```
[1, [2, 3], 4]        ⇒ [1, [2, 3], 4]
[1, (values 2 3), 4] ⇒ [1, 2, 3, 4]
[1, (values), 3]      ⇒ [1, 3]
```

When a template is a struct, it expands into a struct with the same annotations. The struct-template’s field names are treated as literals, and field values are treated as sub-templates, and the output struct contains the given names and their associated sub-template expansions.

Field-value sub-templates MAY produce multiple values. When a sub-template produces more than one result, then the output struct will have more than one field with the same name. When a sub-template produces no results, then nothing is added to the output.

```
{a:(values 1 2)} ⇒ {a:1, a:2}   // or, equivalently, {a:2, a:1}
{f:(values)}     ⇒ {}
```

9.1.4 S-expressions are Operator Invocations

The template language uses S-expressions to denote operations using Lisp-style prefix notation. The first element of the S-expression must be a symbol that identifies the operator, and the meaning of subsequent elements depends on the operator.

Operators come in two varieties: special forms and macro invocations.

9.2 Special Forms

Special forms are operators that cannot be expressed as macros, because some parts of their syntax are not recursively-expanded templates, as all macro arguments are.

We use **`bold monospace`** when naming these special forms, to distinguish them from macro names.

In the descriptions below, *`template`* subforms accept any template-language form. In all such cases, sub-templates are expanded only when indicated.

9.2.1 Preventing Evaluation

9.2.1.1 literal

```
(literal datum)
```

Produces *datum* as-is, preventing the operand from being evaluated as a template.

For example, `(literal [1, (values 2 3), 4])` produces `[1, (values 2 3), 4]`; both the list and the S-expression are treated as literal, constant data, not as template expressions to be expanded.

9.2.2 Conditionals

These special forms allow output to vary based on whether a template produces zero, one, or more values.

9.2.2.1 `if_void`

```
(if_void template_cond template_then template_else)
```

Evaluates templates conditionally based on the cardinality of a stream.

The $template_{cond}$ is expanded to see if it produces any values. If and only if it produces no values, then $template_{then}$ is expanded and its results returned. Otherwise, $template_{else}$ is expanded and its results returned.

9.2.2.2 `if_single`

```
(if_single template_cond template_then template_else)
```

Like **`if_void`**, but expands $template_{then}$ if and only if $template_{cond}$ produces exactly one value, otherwise expands $template_{else}$.

9.2.2.3 `if_many`

```
(if_many template_cond template_then template_else)
```

Like **`if_void`**, but expands $template_{then}$ if and only if $template_{cond}$ produces more than one value, otherwise expands $template_{else}$.

```
(macro decimal_constraint
    [(precision int*), (exponent int*)]
    {
        precision: (if_many precision range::[precision] precision),
        exponent:  (if_many exponent  range::[exponent]  exponent),
    })
```

```
(:decimal_constraint (3) (-1))      ⇒ { precision: 3, exponent: -1 }
(:decimal_constraint (1 5) (-5 0)) ⇒ { precision: range::[1, 5],
                                       exponent: range::[-5, 0] }
(:decimal_constraint (:) (3 max))   ⇒ { exponent: range::[3, max] }
(:decimal_constraint (1) (:))       ⇒ { precision: 1 }
```

9.2.3 Mapping

These special forms produce repeated output mapped across elements of a stream.

9.2.3.1 `for`

```
(for [(id template_in), ...] template_body)
```

Iteratively expands the $template_{body}$ using individual values from the *in-templates*.

Each iteration takes the next value from each $template_{in}$ stream; iteration stops when any stream ends. Local variables are created for each identifier *id*, bound to the current value from their stream. The $template_{body}$ is then expanded in that environment, and iteration proceeds. The result of the **`for`** expression is the concatenated results of the body expansions.

Note
The termination rule is under discussion; see https://github.com/amazon-ion/ion-docs/issues/201

9.3 Macro Invocation

A macro definition can express its output in terms of other macros. Quite often, these will be macros provided by the Ion implementation, but they can also be acquired from other modules.

The S-expression syntax for macro invocation is similar to that of E-expressions. When a template is an S-expression and the first element is not the name of a special form, that element must instead be a *macro-ref* and the template denotes a macro invocation. There are multiple sources of macros: the defining module's internal environment (which is being incrementally extended with each definition), and the exported macros of modules loaded by the enclosing module or `$ion_encoding` directive.

The remaining elements of the S-expression are subforms that denote the inputs to the macro. These use normal Ion notation, but what's syntactically acceptable is defined by the macro's signature.

The number of such subforms (that is, the invocation's actual arity) must be equal to or greater than the macro's minimum arity, and at most its maximum arity, when one exists. In other words, an invocation must contain one subform for each required parameter, followed by optional subforms for the remaining optional parameters.

Within an invocation expression, the syntax of each subform is defined first by its parameter's grouping form, then its base type:

- The subform for a simple parameter must match the base type below.
- The subform for a grouped parameter must be a list containing elements that each match the base type.
- A rest parameter captures all remaining subforms of the invocation, each of which must match the base type.

The base types match as follows:

- For tagged types, the subform may be any template that produces acceptable values.
- For primitive types, the subform may be any template that produces values accepted by the corresponding concrete type.
- For macro types, the subform must be an S-expression containing subforms acceptable to that macro's signature. These are implicit invocations of the macro, and the macro name cannot be provided explicitly.

TODO Allow macro invocations where grouping list is expected?

TODO Clarify when/where range checks are applied for fixed-width types.

TODO Examples

9.4 Type Checking

TODO

9.5 Error Handling

TODO

Chapter 10

Ion 1.1 Binary Encoding

10.1 Encoding Primitives

10.1.1 `FlexUInt`

A variable-length unsigned integer.

The bytes of a `FlexUInts` are written in little-endian byte order. This means that the first bytes will contain the `FlexUInt`'s least significant bits.

The least significant bits in the `FlexUInt` indicate the number of bytes that were used to encode the integer. If a `FlexUInt` is *N* bytes long, its *N-1* least significant bits will be `0`; a terminal `1` bit will be in the next most significant position. All bits that are more significant than the terminal `1` represent the magnitude of the `FlexUInt`.

Figure 1: FlexUInt encoding of 14

```
              ┌──── Lowest bit is 1 (end), indicating
              │     this is the only byte.
0 0 0 1 1 1 0 1
└─────┬─────┘
unsigned int 14
```

Figure 2: FlexUInt encoding of 729

```
             ┌──── There's 1 zero in the least significant bits, so this
             │     integer is two bytes wide.
            ┌┴┐
0 1 1 0 0 1 1 0  0 0 0 0 1 0 1 1
└────┬────┘      └──────┬──────┘
lowest 6 bits    highest 8 bits
of the unsigned  of the unsigned
integer          integer
```

Figure 3: FlexUInt encoding of 21,043

```
            ┌───── There are 2 zeros in the least significant bits, so this
            │      integer is three bytes wide.
          ┌─┴─┐
1 0 0 1 1 1 0 0  1 0 0 1 0 0 0 1  0 0 0 0 0 0 1 0
└───┬───┘        └──────┬──────┘  └──────┬──────┘
lowest 6 bits    next 8 bits of   highest 8 bits
of the unsigned  the unsigned     of the unsigned
integer          integer          integer
```

10.1.2 `FlexInt`

A variable-length signed integer.

From an encoding perspective, `FlexInts` are structurally similar to a `FlexUInt` (described above). Both encode their bytes using little-endian byte order, and both use the count of least-significant zero bits to indicate how many bytes were used to encode the integer. They differ in the *interpretation* of their bits; while a `FlexUInt`'s bits are unsigned, a `FlexInt`'s bits are encoded using two's complement notation.

Tip

An implementation could choose to read a `FlexInt` by instead reading a `FlexUInt` and then reinterpreting its bits as two's complement.

Figure 4: FlexInt encoding of 14

```
              ┌──── Lowest bit is 1 (end), indicating
              │     this is the only byte.
0 0 0 1 1 1 0 1
└─────┬─────┘
 2's comp. 14
```

Figure 5: FlexInt encoding of -14

```
              ┌──── Lowest bit is 1 (end), indicating
              │     this is the only byte.
1 1 1 0 0 1 0 1
└─────┬─────┘
 2's comp. -14
```

Figure 6: FlexInt encoding of 729

```
             ┌──── There's 1 zero in the least significant bits, so this
             │     integer is two bytes wide.
            ┌┴┐
0 1 1 0 0 1 1 0  0 0 0 0 1 0 1 1
└────┬────┘      └──────┬──────┘
lowest 6 bits    highest 8 bits
of the 2's       of the 2's
comp. integer    comp. integer
```

Figure 7: FlexInt encoding of -729

```
             ┌──── There's 1 zero in the least significant bits, so this
             │     integer is two bytes wide.
            ┌┴┐
1 0 0 1 1 1 1 0  1 1 1 1 0 1 0 0
└────┬────┘      └──────┬──────┘
lowest 6 bits    highest 8 bits
of the 2's       of the 2's
comp. integer    comp. integer
```

10.1.3 `FixedUInt`

A fixed-width, little-endian, unsigned integer whose length is inferred from the context in which it appears.

Figure 8: FixedUInt encoding of 3,954,261


```
0 1 0 1 0 1 0 1  0 1 0 1 0 1 1 0  0 0 1 1 1 1 0 0
└──────┬──────┘  └──────┬──────┘  └──────┬──────┘
lowest 8 bits    next 8 bits of   highest 8 bits
of the unsigned  the unsigned     of the unsigned
integer          integer          integer
```

10.1.4 `FixedInt`

A fixed-width, little-endian, signed integer whose length is known from the context in which it appears. Its bytes are interpreted as two's complement.

Figure 9: FixedInt encoding of -3,954,261

```
1 0 1 0 1 0 1 1  1 0 1 0 1 0 0 1  1 1 0 0 0 0 1 1
└──────┬──────┘  └──────┬──────┘  └──────┬──────┘
lowest 8 bits    next 8 bits of   highest 8 bits
of the 2's       the 2's comp.    of the 2's comp.
comp. integer    integer          integer
```

10.1.5 `FlexSym`

A variable-length symbol token whose UTF-8 bytes can be inline, found in the symbol table, or derived from a macro expansion.

A `FlexSym` begins with a FlexInt; once this integer has been read, we can evaluate it to determine how to proceed. If the FlexInt is:

- **greater than zero**, it represents a symbol ID. The symbol's associated text can be found in the local symbol table. No more bytes follow.
- **less than zero**, its absolute value represents a number of UTF-8 bytes that follow the `FlexInt`. These bytes represent the symbol's text.
- **exactly zero**, another byte follows that is an opcode. The `FlexSym` parser is not responsible for evaluating this opcode, only returning it—the caller will decide whether the opcode is legal in the current context. Example usages of the opcode include:
  – Representing SID `$0` as `0x70`. (See: Strings)
  – Representing the empty string (`""`) as `0x80`. (See: Symbols with inline text)
  – When used to encode a struct field name, the opcode can invoke a macro that will evaluate to a struct whose key/value pairs are spliced into the parent struct (TODO: Link)
  – In a delimited struct, terminating the sequence of `(field name, value)` pairs with `0xF0`.

Figure 10: FlexSym encoding of symbol ID $10

```
              ┌─── The leading FlexInt ends in a `1`,
              │    no more FlexInt bytes follow.
              │
0 0 0 1 0 1 0 1
└─────┬─────┘
 2's comp.
 positive 10
```

Figure 11: FlexSym encoding of symbol text 'hello'


```
                ┌─── The leading FlexInt ends in a `1`,
                │    no more FlexInt bytes follow.
                │      h         e        l        l        o
1 1 1 1 0 1 1 1  01101000  01100101 01101100 01101100 01101111
└──────┬─────┘   └──────────────────────┬─────────────────────┘
  2's comp.                5-byte UTF-8 encoded "hello"
  negative 5
```

Figure 12: FlexSym encoding of '' (empty text) using an opcode

```
                ┌─── The leading FlexInt ends in a `1`,
                │    no more FlexInt bytes follow.
                │
0 0 0 0 0 0 0 1  1110000
└──────┬─────┘   └──┬──┘
  2's comp.      opcode 0x70:
  zero           empty symbol
```

10.2 Opcodes

An *opcode* is a 1-byte FixedUInt that tells the reader what the next expression represents and how the bytes that follow should be interpreted.

The meanings of each opcode are organized loosely by their high and low nibbles.

High nibble	Low nibble	Meaning
`0x0_` to `0x3_`	`0-F`	E-expression with the address in the opcode
`0x4_`	`0-F`	E-expression with the address as a trailing `FlexUInt`
`0x5_`	`0-8`	Integers up to 8 bytes wide
	`9`	*Reserved*
	`A-D`	Floats
	`E-F`	Booleans
`0x6_`	`0-F`	Decimals
`0x7_`	`0-F`	Timestamps
`0x8_`	`0-F`	Strings
`0x9_`	`0-F`	Symbols with inline text
`0xA_`	`0-F`	Lists
`0xB_`	`0-F`	S-expressions
`0xC_`	`0`	Empty struct
	`1`	*Reserved*
	`2-F`	Structs with symbol address field names
`0xD_`	`0-1`	*Reserved*
	`2-F`	Structs with `FlexSym` field names
`0xE_`	`0`	Ion version marker
	`1-3`	Symbols with symbol address
	`4-6`	Annotations with symbol address
	`7-9`	Annotations with `FlexSym` text
	`A`	`null.null`
	`B`	Typed nulls
	`C-D`	NOP
	`E`	*Reserved*
	`F`	System macro invocation
`0xF_`	`0`	Delimited container end
	`1`	Delimited list start
	`2`	Delimited S-expression start

High nibble	Low nibble	Meaning
	3	Delimited struct with `FlexSym` field names start
	4	Variable length prefixed macro invocation
	5	Variable length integer
	6	Variable length decimal
	7	Variable length, long-form timestamp
	8	Variable length string
	9	Variable length symbol encoded as `FlexSym`
	A	Variable length list
	B	Variable length S-expression
	C	Variable length struct with symbol address field names
	D	Variable length struct with `FlexSym` field names
	E	Variable length blob
	F	Variable length clob

10.3 Encoding Expressions

10.3.1 E-expression With the Address in the Opcode

If the value of the opcode is less than `64` (`0x40`), it represents an E-expression invoking the macro at the corresponding *address*—an offset within the local macro table.

Figure 13: Invocation of macro address 7

```
0 0 0 0 0 1 1 1
└──────┬──────┘
  FixedUInt 7
```

Figure 14: invocation of macro address 31

```
0 0 0 1 1 1 1 1
└──────┬──────┘
  FixedUInt 31
```

Note that the opcode alone tells us which macro is being invoked, but it does not supply enough information for the reader to parse any arguments that may follow. The parsing of arguments is described in detail in the section *Macro calling conventions*. (TODO: Link)

10.3.2 E-expression With the Address as a Trailing `FlexUInt`

While E-expressions invoking macro addresses in the range `[0, 63]` can be encoded in a single byte using E-expressions with the address in the opcode, many applications will benefit from defining more than 64 macros.

If the high nibble of the opcode is `0x4_`, then the low nibble represents the four least significant bits of the macro address. A FlexUInt follows that contains the remaining, more significant bits.

Because the first 64 macro addresses can already be encoded using high nibbles `0` to `3`, the decoded value is biased by 64. (That is: the reader must add 64 to the decoded value. If the decoded value is `0`, the macro address that it represents is `64`.)

Because the address is encoded using a `FlexUInt`, there is no (theoretical) limit to the number of addresses that can be invoked. However, larger addresses require more bytes to encode. The following table shows the number of bytes needed to encode invocations of macro addresses in various ranges.

Address range	Bytes needed	Magnitude bits available
0 to 63	1	6
64 to 2,112	2	11

Address range	Bytes needed	Magnitude bits available
2,113 to 262,208	3	18
262,209 to 33,554,432	4	25

Figure 15: Invocation of macro address 131

```
                                          ┌──── The address FlexUInt ends in a ‘1‘,
                                          │     no more FlexUInt bytes follow.
                                          │
0 1 0 0 0 0 1 1  0 0 0 0 1 0 0 1
└──┬──┘ └──┬──┘  └──────┬──────┘
   │       │            └─────────────── FlexUInt containing the 7 most
   │       └─── 4 least significant      significant bits of the macro
opcode high     bits of the macro        address
nibble 4        address

Magnitude bits: 0000100_0011
Decoded value : 67
Biased value  : 131
```

Figure 16: Invocation of macro address 1,211

```
                                          ┌──── The address FlexUInt ends in a ‘1‘,
                                          │     no more FlexUInt bytes follow.
                                          │
0 1 0 0 1 0 1 1  1 0 0 0 1 1 1 1
└──┬──┘ └──┬──┘  └──────┬──────┘
   │       │            └─────────────── FlexUInt containing the 7 most
   │       └─── 4 least significant      significant bits of the macro
opcode high     bits of the macro        address
nibble 4        address

Magnitude bits: 1000111_1011
Decoded value : 1,147
Biased value  : 1,211
```

Figure 17: Invocation of macro address 71,376

```
                                        ┌──── The address FlexUInt ends in ‘10‘; the zero in the least ↩
                                        │    significant
                                        │    bits indicates that one more FlexUInt byte follows.
                                       ┌┴┐
0 1 0 0 0 0 0 0  1 0 1 0 0 1 1 0  0 1 0 0 0 1 0 1
└──┬──┘ └──┬──┘  └──────┬──────┘  └──────┬──────┘
   │       │            │                └──────────── the 8 most significant bits
   │       │            │                              of the macro address
   │       │            │
   │       │            └─────────────── FlexUInt containing the next 7 most
   │       └─── 4 least significant      significant bits of the macro
opcode high     bits of the macro        address
nibble 4        address

Magnitude bits: 01000101_101001_0000
Decoded value : 71,312
Biased value  : 71,376
```

Note
From this point on in the document, example encodings are given in hexadecimal notation.

10.4 Booleans

`0x5E` represents boolean `true`, while `0x5F` represents boolean `false`.

`0xEB 0x00` represents `null.bool`.

Figure 18: Encoding of boolean true

```
5E
```

Figure 19: Encoding of boolean false

```
5F
```

Figure 20: Encoding of null.bool

```
┌──── Opcode 0xEB indicates a typed null; a byte follows specifying the type
│  ┌─── Null type: boolean
│  │
EB 00
```

10.5 Numbers

10.5.1 Integers

Opcodes in the range `0x50` to `0x58` represent an integer. The opcode is followed by a FixedInt that represents the integer value. The low nibble of the opcode (`0x_0` to `0x_8`) indicates the size of the `FixedInt`. Opcode `0x50` represents integer `0`; no more bytes follow.

Integers that require more than 8 bytes are encoded using the variable-length integer opcode `0xF5`, followed by a FlexUInt indicating how many bytes of representation data follow.

`0xEB 0x01` represents `null.int`.

Figure 21: Encoding of integer 0

```
┌──── Opcode in 50-58 range indicates integer
│┌─── Low nibble 0 indicates
││    no more bytes follow.
50
```

Figure 22: Encoding of integer 17

```
┌──── Opcode in 50-58 range indicates integer
│┌─── Low nibble 1 indicates
││    a single byte follows.
51 11
    └── FixedInt 17
```

Figure 23: Encoding of integer -944

```
┌──── Opcode in 50-58 range indicates integer
│┌─── Low nibble 2 indicates
││    that two bytes follow.
52 50 FC
   └─┬─┘
FixedInt -944
```

Figure 24: Encoding of integer -944

```
┌──── Opcode F5 indicates a variable-length integer, FlexUInt length follows
│   ┌─── FlexUInt 2; a 2-byte FixedInt follows
│   │
F5 05 50 FC
      └─┬─┘
   FixedInt -944
```

Figure 25: Encoding of null.int

```
┌──── Opcode 0xEB indicates a typed null; a byte follows specifying the type
│  ┌─── Null type: integer
│  │
EB 01
```

10.5.2 Floats

Float values are encoded using the IEEE-754 specification, and can be serialized in four sizes:

- 0 bits (0 bytes), representing the value 0e0 and indicated by opcode `0x5A`
- 16 bits (2 bytes, half precision), indicated by opcode `0x5B`
- 32 bits (4 bytes, single precision), indicated by opcode `0x5C`
- 64 bits (8 bytes, double precision), indicated by opcode `0x5D`

Note that in the Ion data model, float values are always 64 bits. However, if a value can be losslessly serialized in fewer than 64 bits, Ion implementations may choose to do so.

`0xEB 0x02` represents `null.float`.

Figure 26: Encoding of float 0e0

```
┌──── Opcode in range 5A-5D indicates a float
│┌─── Low nibble A indicates
││    a 0-length float; 0e0
5A
```

Figure 27: Encoding of float 3.14e0

```
┌──── Opcode in range 5A-5D indicates a float
│┌─── Low nibble B indicates a 2-byte float
││
5B 42 47
   └─┬─┘
half-precision 3.14
```

Figure 28: Encoding of float 3.1415927e0

```
┌──── Opcode in range 5A-5D indicates a float
│┌─── Low nibble C indicates a 4-byte,
││    single-precision value.
5C 40 49 0F DB
   └────┬────┘
single-precision 3.1415927
```

Figure 29: Encoding of float 3.141592653589793e0

```
┌──── Opcode in range 5A-5D indicates a float
│┌─── Low nibble D indicates an 8-byte,
││    double-precision value.
5D 40 09 21 FB 54 44 2D 18
   └───────────┬──────────┘
double-precision 3.141592653589793
```

Figure 30: Encoding of null.float

```
┌──── Opcode 0xEB indicates a typed null; a byte follows specifying the type
│  ┌─── Null type: float
│  │
EB 02
```

10.5.3 Decimals

If an opcode has a high nibble of 0x6_, it represents a decimal. Low nibble values 0x_E and below indicate the number of trailing bytes used to encode the decimal.

The body of the decimal is encoded as a FlexInt representing its coefficient, followed by a FixedInt representing its exponent. The width of the exponent is the total length of the decimal encoding minus the length of the coefficient. It is possible for the exponent to have a width of zero, indicating an exponent of 0.

Decimal values that require more than 14 bytes can be encoded using the variable-length decimal opcode: 0xF6.

A decimal with a coefficient of -0 (which cannot be encoded as a FlexInt) is encoded using opcode 6F. The opcode is followed by a FlexInt representing the exponent.

0xEB 0x03 represents null.decimal.

Figure 31: Encoding of decimal 0d0

```
┌──── Opcode in range 60-6F indicates a decimal
│┌─── Low nibble 0 indicates a zero-byte
││    decimal; 0d0
60
```

Figure 32: Encoding of decimal 7d0

```
┌──── Opcode in range 60-6F indicates a decimal
│┌─── Low nibble 1 indicates a 1-byte decimal
││
61 0F
   └─── Coefficient: FlexInt 7; no more bytes follow, so exponent is implicitly 0
```

Figure 33: Encoding of decimal `1.27`

```
┌──── Opcode in range 60-6F indicates a decimal
│┌─── Low nibble 3 indicates a 3-byte decimal
││
63 FD 01 FE
   └─┬─┘ └─── Exponent: 1-byte FixedInt -2
     └────── Coefficient: FlexInt 127
```

Figure 34: Variable-length encoding of decimal 1.27

```
┌──── Opcode F6 indicates a variable-length decimal
│
F6 07 FD 01 FE
   │  └─┬─┘ └─── Exponent: 1-byte FixedInt -2
   │    └────── Coefficient: FlexInt 127
   └───────── Decimal length: FlexUInt 3
```


Figure 35: Encoding of -0d3, which has a coefficient of negative zero

```
┌──── Opcode 6F indicates a variable-length decimal with a coefficient of -0
│
6F 07
   └────── Exponent: FlexInt 3
```

Figure 36: Encoding of null.decimal

```
┌──── Opcode 0xEB indicates a typed null; a byte follows specifying the type
│  ┌─── Null type: decimal
│  │
EB 03
```

10.6 Timestamps

Note
In Ion 1.0, text timestamp fields were encoded using the local time while binary timestamp fields were encoded using UTC time. This required applications to perform conversion logic when transcribing from one format to the other. **In Ion 1.1, all binary timestamp fields are encoded in local time.**

Timestamps have two encodings:

Short-form timestamps A compact representation optimized for the most commonly used precisions and date ranges.

Long-form timestamps A less compact representation capable of representing any timestamp in the Ion data model.

`0xEB x04` represents `null.timestamp`.

Figure 37: Encoding of null.timestamp

```
┌──── Opcode 0xEB indicates a typed null; a byte follows specifying the type
│  ┌─── Null type: timestamp
│  │
EB 04
```

10.6.1 Short-form Timestamp

If an opcode has a high nibble of `0x7_`, it represents a short-form timestamp. This encoding focuses on making the most common timestamp precisions and ranges the most compact; less common precisions can still be expressed via the variable-length long form timestamp encoding.

Timestamps may be encoded using the short form if they meet all of the following conditions:

The year is between 1970 and 2097. The year subfield is encoded as the number of years since 1970. 7 bits are dedicated to representing the biased year, allowing timestamps through the year 2097 to be encoded in this form.

The local offset is either UTC, unknown, or falls between `-14:00` to `+14:00` and is divisible by 15 minutes. 7 bits are dedicated to representing the local offset as the number of quarter hours from -56 (that is: offset `-14:00`). The value `0b1111111` indicates an unknown offset. At the time of this writing (2023-05T), all real-world offsets fall between `-12:00` and `+14:00` and are multiples of 15 minutes.

The fractional seconds are a common precision. The timestamp's fractional second precision (if present) is either 3 digits (milliseconds), 6 digits (microseconds), or 9 digits (nanoseconds).

10.6.1.1 Opcodes by precision and offset

Each opcode with a high nibble of `0x7_` indicates a different precision and offset encoding pair.

Opcode	Precision	Serialized size in bytes*	Offset encoding
0x70	Year	1	Implicitly *Unknown offset*
0x71	Month	2	
0x72	Day	2	
0x73	Hour and minutes	4	1 bit to indicate *UTC* or *Unknown Offset*
0x74	Seconds	5	
0x75	Milliseconds	6	
0x76	Microseconds	7	
0x77	Nanoseconds	8	
0x78	Hour and minutes	5	7 bits to represent a known offset. This encoding can also represent *UTC* and *Unknown Offset*, though it is less compact than opcodes `0x73`-`0x77` above.
0x79	Seconds	5	
0x7A	Milliseconds	7	
0x7B	Microseconds	8	
0x7C	Nanoseconds	9	
0x7D	*Reserved*		
0x7E			
0x7F			

** Serialized size in bytes does not include the opcode.*

The body of a short-form timestamp is encoded as a `FixedUInt` of the size specified by the opcode. This integer is then partitioned into bit-fields representing the timestamp's subfields. Note that endianness does not apply here because the bit-fields are defined over the body interpreted as an *integer*.

The following letters to are used to denote bits in each subfield in diagrams that follow. Subfields occur in the same order in all encoding variants, and consume the same number of bits, with the exception of the fractional bits, which consume only enough bits to represent the fractional precision supported by the opcode being used.

Letter code	Number of bits	Subfield
Y	7	Year
M	4	Month
D	5	Day
H	5	Hour
m	6	Minute
o	7	Offset
U	1	Unknown or UTC offset
s	6	Second
f	10 (ms) 20 (μs) 30 (ns)	Fractional second
.	n/a	Unused

We will denote the timestamp encoding as follows with each byte ordered vertically from top to bottom. The respective bits are denoted using the letter codes defined in the table above.

```
          7       0 <--- bit position
          |       |
         +=========+
byte 0   |  0xNN   | <-- hex notation for constants like opcodes
         +=========+ <-- boundary between encoding primitives (e.g., opcode/`FlexUInt`)
     1   |nnnn:nnnn| <-- bits denoted with a `:` as a delimeter to aid in reading
         +---------+ <-- octet boundary within an encoding primitive
```

```
          ...
          +---------+
     N    |nnnn:nnnn|
          +=========+
```

The bytes are read from top to bottom (least significant to most significant), while the bits within each byte should be read from right to left (also least significant to most significant.)

Note

While this encoding may complicate human reading, it guarantees that the timestamp's subfields (`year`, `month`, etc.) occupy the same bit contiguous indexes regardless of how many bytes there are overall. (The last subfield, `fractional_seconds`, always begins at the same bit index when present, but can vary in length according to the precision.) This arrangement allows processors to read the Little-Endian bytes into an integer and then mask the appropriate bit ranges to access the subfields.

Figure 38: Encoding of a timestamp with year precision

```
          +=========+
byte 0    |  0x70   |
          +=========+
     1    |.YYY:YYYY|
          +=========+
```

Figure 39: Encoding of a timestamp with month precision

```
          +=========+
byte 0    |  0x71   |
          +=========+
     1    |MYYY:YYYY|
          +---------+
     2    |....:.MMM|
          +=========+
```

Figure 40: Encoding of a timestamp with day precision

```
          +=========+
byte 0    |  0x72   |
          +=========+
     1    |MYYY:YYYY|
          +---------+
     2    |DDDD:DMMM|
          +=========+
```

Figure 41: Encoding of a timestamp with hour-and-minutes precision at UTC or unknown offset

```
          +=========+
byte 0    |  0x73   |
          +=========+
     1    |MYYY:YYYY|
          +---------+
     2    |DDDD:DMMM|
          +---------+
     3    |mmmH:HHHH|
          +---------+
     4    |....:Ummm|
          +=========+
```

Figure 42: Encoding of a timestamp with seconds precision at UTC or unknown offset


```
         +=========+
byte 0   |  0x74   |
         +=========+
     1   |MYYY:YYYY|
         +---------+
     2   |DDDD:DMMM|
         +---------+
     3   |mmmH:HHHH|
         +---------+
     4   |ssss:Ummm|
         +---------+
     5   |....:..ss|
         +=========+
```

Figure 43: Encoding of a timestamp with milliseconds precision at UTC or unknown offset

```
         +=========+
byte 0   |  0x75   |
         +=========+
     1   |MYYY:YYYY|
         +---------+
     2   |DDDD:DMMM|
         +---------+
     3   |mmmH:HHHH|
         +---------+
     4   |ssss:Ummm|
         +---------+
     5   |ffff:ffss|
         +---------+
     6   |....:ffff|
         +=========+
```

Figure 44: Encoding of a timestamp with microseconds precision at UTC or unknown offset

```
         +=========+
byte 0   |  0x76   |
         +=========+
     1   |MYYY:YYYY|
         +---------+
     2   |DDDD:DMMM|
         +---------+
     3   |mmmH:HHHH|
         +---------+
     4   |ssss:Ummm|
         +---------+
     5   |ffff:ffss|
         +---------+
     6   |ffff:ffff|
         +---------+
     7   |..ff:ffff|
         +=========+
```

Figure 45: Encoding of a timestamp with nanoseconds precision at UTC or unknown offset

```
         +=========+
byte 0   |  0x77   |
         +=========+
     1   |MYYY:YYYY|
         +---------+
     2   |DDDD:DMMM|
         +---------+
```

```
     3   |mmmH:HHHH|
         +---------+
     4   |ssss:Ummm|
         +---------+
     5   |ffff:ffss|
         +---------+
     6   |ffff:ffff|
         +---------+
     7   |ffff:ffff|
         +---------+
     8   |ffff:ffff|
         +=========+
```

Figure 46: Encoding of a timestamp with hour-and-minutes precision at known offset

```
         +=========+
byte 0   |  0x78   |
         +=========+
     1   |MYYY:YYYY|
         +---------+
     2   |DDDD:DMMM|
         +---------+
     3   |mmmH:HHHH|
         +---------+
     4   |oooo:ommm|
         +---------+
     5   |....:..oo|
         +=========+
```

Figure 47: Encoding of a timestamp with seconds precision at known offset

```
         +=========+
byte 0   |  0x79   |
         +=========+
     1   |MYYY:YYYY|
         +---------+
     2   |DDDD:DMMM|
         +---------+
     3   |mmmH:HHHH|
         +---------+
     4   |oooo:ommm|
         +---------+
     5   |ssss:ssoo|
         +=========+
```

Figure 48: Encoding of a timestamp with milliseconds precision at known offset

```
         +=========+
byte 0   |  0x7A   |
         +=========+
     1   |MYYY:YYYY|
         +---------+
     2   |DDDD:DMMM|
         +---------+
     3   |mmmH:HHHH|
         +---------+
     4   |oooo:ommm|
         +---------+
     5   |ssss:ssoo|
         +---------+
     6   |ffff:ffff|
         +---------+
```

```
      7   |....:..ff|
          +=========+
```

Figure 49: Encoding of a timestamp with microseconds precision at known offset

```
          +=========+
byte 0    |   0x7B  |
          +=========+
      1   |MYYY:YYYY|
          +---------+
      2   |DDDD:DMMM|
          +---------+
      3   |mmmH:HHHH|
          +---------+
      4   |oooo:ommm|
          +---------+
      5   |ssss:ssoo|
          +---------+
      6   |ffff:ffff|
          +---------+
      7   |ffff:ffff|
          +---------+
      8   |....:ffff|
          +=========+
```

Figure 50: Encoding of a timestamp with nanoseconds precision at known offset

```
          +=========+
byte 0    |   0x7C  |
          +=========+
      1   |MYYY:YYYY|
          +---------+
      2   |DDDD:DMMM|
          +---------+
      3   |mmmH:HHHH|
          +---------+
      4   |oooo:ommm|
          +---------+
      5   |ssss:ssoo|
          +---------+
      6   |ffff:ffff|
          +---------+
      7   |ffff:ffff|
          +---------+
      8   |ffff:ffff|
          +---------+
      9   |..ff:ffff|
          +=========+
```

Warning
Opcodes `0x7D`, `0x7E`, and `7F` are illegal; they are reserved for future use.

10.6.2 Long-form Timestamp

Unlike the Short-form timestamp encoding, which is limited to encoding timestamps in the most commonly referenced timestamp ranges and precisions for which it optimizes, the long-form timestamp encoding is capable of representing any valid timestamp.

The long form begins with opcode `0xF7`. A FlexUInt follows indicating the number of bytes that were needed to represent the timestamp. The encoding consumes the minimum number of bytes required to represent the timestamp. The declared length can be mapped to the timestamp's precision as follows:

Length	Corresponding precision
0	*Illegal*
1	*Illegal*
2	Year
3	Month or Day (see below)
4	*Illegal; the hour cannot be specified without also specifying minutes*
5	*Illegal*
6	Minutes
7	Seconds
8 or more	Fractional seconds

Unlike the short-form encoding, the long-form encoding reserves:

- **14 bits for the year (`Y`)**, which is not biased.
- **12 bits for the offset**, which counts the number of minutes (not quarter-hours) from -1440 (that is: `-24:00`). An offset value of `0b111111111111` indicates an unknown offset.

Similar to short-form timestamps, with the exception of representing the fractional seconds, the components of the timestamp are encoded as bit-fields on a `FixedUInt` that corresponds to the length that followed the opcode.

If the timestamp's overall length is greater than or equal to `8`, the `FixedUInt` part of the timestamp is `7` bytes and the remaining bytes are used to encode fractional seconds. The fractional seconds are encoded as a `(coefficient, scale)` pair, which is *similar* to a decimal. The primary difference is that the **scale** represents a negative **exponent** because it is illegal for the fractional seconds value to be greater than or equal to `1.0` or less than `0.0`. The coefficient is encoded as a `FlexUInt` (instead of `FlexInt`) to prevent the encoding of fractional seconds less than `0.0`. The scale is encoded as a `FixedUInt` (instead of `FixedInt`) to discourage the encoding of decimal numbers greater than `1.0`. Note that validation is still required; namely:

- A scale value of `0` is illegal, as that would result in a fractional seconds greater than `1.0` (a whole second).
- If `coefficient * 10^-scale > 1.0`, that `(coefficient, scale)` pair is illegal.

If the timestamp's length is `3`, the most significant bit in the final byte (`h`) is a flag that indicates month (`0`) or day (`1`) precision. If the timestamp's length is greater than `3`, the (`h`) bit is treated as the least-significant bit of the hour (`H`) bits.

Figure 51: Encoding of the body of a long-form timestamp

```
          +=========+
byte 0    |YYYY:YYYY|
          +=========+
     1    |MMYY:YYYY|
          +---------+
     2    |hDDD:DDMM|
          +---------+
     3    |mmmm:HHHH|
          +---------+
     4    |oooo:oomm|
          +---------+
     5    |ssoo:oooo|
          +---------+
     6    |....:ssss|
          +=========+
     7    |FlexUInt | <-- coefficient of the fractional seconds
          +---------+
          ...
```



```
     +=========+
N    |FixedUInt| <-- scale of the fractional seconds
     +---------+
     ...
```

10.7 Text

10.7.1 Strings

If the high nibble of the opcode is `0x8_`, it represents a string. The low nibble of the opcode indicates how many UTF-8 bytes follow. Opcode `0x80` represents a string with empty text (`""`).

Strings longer than 15 bytes can be encoded with the `F8` opcode, which takes a FlexUInt-encoded length after the opcode.

`0xEB x05` represents `null.string`.

Figure 52: Encoding of the empty string, ""

```
┌──── Opcode in range 80-8F indicates a string
│┌─── Low nibble 0 indicates that no UTF-8 bytes follow
80
```

Figure 53: Encoding of a 14-byte string

```
┌──── Opcode in range 80-8F indicates a string
│┌─── Low nibble E indicates that 14 UTF-8 bytes follow
││  f  o  u  r  t  e  e  n     b  y  t  e  s
8E 66 6F 75 72 74 65 65 6E 20 62 79 74 65 73
   └────────────────────┬────────────────────┘
                   UTF-8 bytes
```

Figure 54: Encoding of a 24-byte string

```
┌──── Opcode F8 indicates a variable-length string
│  ┌─── Length: FlexUInt 24
│  │  v  a  r  i  a  b  l  e     l  e  n  g  t  h     e  n  c  o  d  i  n  g
F8 31 76 61 72 69 61 62 6C 65 20 6C 65 6E 67 74 68 20 65 6E 63 6f 64 69 6E 67
      └──────────────────────────────────┬──────────────────────────────────────┘
                                    UTF-8 bytes
```

Figure 55: Encoding of null.string

```
┌──── Opcode 0xEB indicates a typed null; a byte follows specifying the type
│  ┌─── Null type: string
│  │
EB 05
```

10.7.2 Symbols With Inline Text

If the high nibble of the opcode is `0x9_`, it represents a symbol whose text follows the opcode. The low nibble of the opcode indicates how many UTF-8 bytes follow. Opcode `0x90` represents a symbol with empty text (″).

`0xEB x06` represents `null.symbol`.

Figure 56: Encoding of a symbol with empty text (”)

```
┌──── Opcode in range 90-9F indicates a symbol with inline text
│┌─── Low nibble 0 indicates that no UTF-8 bytes follow
90
```

Figure 57: Encoding of a symbol with 14 bytes of inline text

```
┌──── Opcode in range 90-9F indicates a symbol with inline text
│ ┌── Low nibble E indicates that 14 UTF-8 bytes follow
│ │  f  o  u  r  t  e  e  n     b  y  t  e  s
9E 66 6F 75 72 74 65 65 6E 20 62 79 74 65 73
   └──────────────────┬──────────────────┘
                  UTF-8 bytes
```

Figure 58: Encoding of a symbol with 24 bytes of inline text

```
┌──── Opcode F9 indicates a variable-length symbol with inline text
│  ┌── Length: FlexUInt 24
│  │  v  a  r  i  a  b  l  e     l  e  n  g  t  h     e  n  c  o  d  i  n  g
F9 31 76 61 72 69 61 62 6C 65 20 6C 65 6E 67 74 68 20 65 6E 63 6f 64 69 6E 67
      └─────────────────────────────────┬─────────────────────────────────┘
                                   UTF-8 bytes
```

Figure 59: Encoding of null.symbol

```
┌──── Opcode 0xEB indicates a typed null; a byte follows specifying the type
│  ┌── Null type: symbol
│  │
EB 06
```

10.7.3 Symbols With a Symbol Address

Symbol values whose text can be found in the local symbol table are encoded using opcodes `0xE1` through `0xE3`:

- `0xE1` represents a symbol whose address in the symbol table (aka its symbol ID) is a 1-byte FixedUInt that follows the opcode.
- `0xE2` represents a symbol whose address in the symbol table is a 2-byte FixedUInt that follows the opcode.
- `0xE3` represents a symbol whose address in the symbol table is a FlexUInt that follows the opcode.

Writers MUST encode a symbol address in the smallest number of bytes possible. For each opcode above, the symbol address that is decoded is biased by the number of addresses that can be encoded in fewer bytes.

Opcode	Symbol address range	Bias
`0xE1`	0 to 255	0
`0xE2`	256 to 65,791	256
`0xE3`	65,792 to infinity	65,792

10.8 Binary Data

10.8.1 Blobs

Opcode `FE` indicates a blob of binary data. A `FlexUInt` follows that represents the blob's byte-length.

`0xEB x07` represents `null.blob`.

Figure 60: Encoding of a blob with 24 bytes of data

```
┌──── Opcode FE indicates a blob, FlexUInt length follows
│   ┌── Length: FlexUInt 24
│   │
FE 31 49 20 61 70 70 6c 61 75 64 20 79 6f 75 72 20 63 75 72 69 6f 73 69 74 79
      └─────────────────────────────────┬─────────────────────────────────┘
                              24 bytes of binary data
```


Figure 61: Encoding of null.blob

```
┌──── Opcode 0xEB indicates a typed null; a byte follows specifying the type
│  ┌─── Null type: blob
│  │
EB 07
```

10.8.2 Clobs

Opcode `FF` indicates a clob—binary character data of an unspecified encoding. A `FlexUInt` follows that represents the clob's byte-length.

`0xEB x08` represents `null.clob`.

Figure 62: Encoding of a clob with 24 bytes of data

```
┌──── Opcode FF indicates a clob, FlexUInt length follows
│   ┌─── Length: FlexUInt 24
│   │
FF 31 49 20 61 70 70 6c 61 75 64 20 79 6f 75 72 20 63 75 72 69 6f 73 69 74 79
      └─────────────────────────────────┬─────────────────────────────────┘
                             24 bytes of binary data
```

Figure 63: Encoding of null.clob

```
┌──── Opcode 0xEB indicates a typed null; a byte follows specifying the type
│  ┌─── Null type: clob
│  │
EB 08
```

10.9 Containers

Each of the container types (list, s-expression, and struct) has both a length-prefixed encoding and a delimited encoding.

The length-prefixed encoding places more burden on the writer, but simplifies reading and enables skipping over uninteresting values in the data stream. In contrast, the delimited encoding is simpler and faster for writers, but requires the reader to visit each child value in turn to skip over the container.

10.9.1 Lists

10.9.1.1 Length-prefixed encoding

An opcode with a high nibble of `0xA_` indicates a length-prefixed list. The lower nibble of the opcode indicates how many bytes were used to encode the child values that the list contains.

If the list's encoded byte-length is too large to be encoded in a nibble, writers may use the `0xFA` opcode to write a variable-length list. The `0xFA` opcode is followed by a FlexUInt that indicates the list's byte length.

`0xEB 0x09` represents `null.list`.

Figure 64: Length-prefixed encoding of an empty list ([])

```
┌──── An Opcode in the range 0xA0-0xAF indicates a list.
│┌─── A low nibble of 0 indicates that the child values of this list took zero bytes to ↩
    encode.
A0
```

Figure 65: Length-prefixed encoding of [1, 2, 3]

```
┌──── An Opcode in the range 0xA0-0xAF indicates a list.
│ ┌── A low nibble of 0 indicates that the child values of this list took zero bytes to ↩
    encode.
A6 51 01 51 02 51 03
   └─┬─┘ └─┬─┘ └─┬─┘
     1     2     3
```

Figure 66: Length-prefixed encoding of ["variable length list"]

```
┌──── Opcode 0xFA indicates a variable-length list. A FlexUInt length follows.
│  ┌───── Length: FlexUInt 22
│  │  ┌────── Opcode 0xF8 indicates a variable-length string. A FlexUInt length follows.
│  │  │  ┌─────── Length: FlexUInt 20
│  │  │  │  v  a  r  i  a  b  l  e     l  e  n  g  t  h     l  i  s  t
FA 2d F8 29 76 61 72 69 61 62 6c 65 20 6c 65 6e 67 74 68 20 6c 69 73 74
      └──────────────────────────────┬──────────────────────────────┘
                            Nested string element
```

Figure 67: Encoding of null.list

```
┌──── Opcode 0xEB indicates a typed null; a byte follows specifying the type
│  ┌─── Null type: list
│  │
EB 09
```

10.9.1.2 Delimited Encoding

Opcode `0xF1` begins a delimited list, while opcode `0xF0` closes the most recently opened delimited container that has not yet been closed.

Figure 68: Delimited encoding of an empty list ([])

```
┌──── Opcode 0xF1 indicates a delimited list
│  ┌─── Opcode 0xF0 indicates the end of the most recently opened container
F1 F0
```

Figure 69: Delimited encoding of [1, 2, 3]

```
┌──── Opcode 0xF1 indicates a delimited list
│                    ┌─── Opcode 0xF0 indicates the end of
│                    │    the most recently opened container
F1 51 01 51 02 51 03 F0
   └─┬─┘ └─┬─┘ └─┬─┘
     1     2     3
```

Figure 70: Delimited encoding of [1, [2], 3]

```
┌──── Opcode 0xF1 indicates a delimited list
│        ┌─── Opcode 0xF1 begins a nested delimited list
│        │        ┌─── Opcode 0xF0 closes the most recently
│        │        │    opened delimited container: the nested list.
│        │        │        ┌─── Opcode 0xF0 closes the most recently opened (and still open ↩
   )
│        │        │        │    delimited container: the outer list.
│        │        │        │
F1 51 01 F1 51 02 F0 51 03 F0
   └─┬─┘    └─┬─┘    └─┬─┘
     1        2        3
```

10.9.2 S-Expressions

S-expressions use the same encodings as lists, but with different opcodes.

Opcode	Encoding
0xB0-0xBF	Length-prefixed S-expression; low nibble of the opcode represents the byte-length.
0xFB	Variable-length prefixed S-expression; a FlexUInt following the opcode represents the byte-length.
0xF2	Starts a delimited S-expression; 0xF0 closes the most recently opened delimited container.

0xEB 0x0A represents null.sexp.

Figure 71: Length-prefixed encoding of an empty S-expression (())

```
┌──── An Opcode in the range 0xB0-0xBF indicates an S-expression.
│ ┌─── A low nibble of 0 indicates that the child values of this S-expression took zero ↩
   bytes to encode.
B0
```

Figure 72: Length-prefixed encoding of (1 2 3)

```
┌──── An Opcode in the range 0xB0-0xBF indicates an S-expression.
│ ┌─── A low nibble of 6 indicates that the child values of this S-expression took six bytes ↩
    to encode.
B6 51 01 51 02 51 03
   └─┬─┘ └─┬─┘ └─┬─┘
     1     2     3
```

Figure 73: Length-prefixed encoding of ("variable length sexp")

```
┌──── Opcode 0xFB indicates a variable-length list. A FlexUInt length follows.
│  ┌───── Length: FlexUInt 22
│  │  ┌───── Opcode 0xF8 indicates a variable-length string. A FlexUInt length follows.
│  │  │  ┌───── Length: FlexUInt 20
│  │  │  │  v  a  r  i  a  b  l  e     l  e  n  g  t  h     s  e  x  p
FB 2D F8 29 76 61 72 69 61 62 6C 65 20 6C 65 6E 67 74 68 20 73 65 78 70
      └────────────────────────────────┬────────────────────────────────┘
                            Nested string element
```

Figure 74: Delimited encoding of an empty S-expression (())

```
┌──── Opcode 0xF2 indicates a delimited S-expression
│  ┌─── Opcode 0xF0 indicates the end of the most recently opened container
F2 F0
```

Figure 75: Delimited encoding of (1 2 3)

```
┌──── Opcode 0xF2 indicates a delimited S-expression
│                    ┌─── Opcode 0xF0 indicates the end of
│                    │    the most recently opened container
F2 51 01 51 02 51 03 F0
   └─┬─┘ └─┬─┘ └─┬─┘
     1     2     3
```

Figure 76: Delimited encoding of (1 (2) 3)

```
┌──── Opcode 0xF2 indicates a delimited S-expression
│        ┌───── Opcode 0xF2 begins a nested delimited S-expression
│        │        ┌─── Opcode 0xF0 closes the most recently
│        │        │    opened delimited container: the nested S-expression.
```

```
│          │          │          ┌─── Opcode 0xF0 closes the most recently opened (and still open ↩
   )
│          │          │          │    delimited container: the outer S-expression.
│          │          │          │
F2 51 01 F2 51 02 F0 51 03 F0
   └─┬─┘      └─┬─┘      └─┬─┘
     1          2          3
```

Figure 77: Encoding of null.sexp

```
┌──── Opcode 0xEB indicates a typed null; a byte follows specifying the type
│  ┌─── Null type: sexp
│  │
EB 0A
```

10.9.3 Structs

Structs have 3 available encodings:

1. Structs with symbol address field names
2. Structs with FlexSym field names
3. Delimited structs with FlexSym field names

`0xEB 0x0B` represents `null.struct`.

Figure 78: Encoding of null.struct

```
┌──── Opcode 0xEB indicates a typed null; a byte follows specifying the type
│  ┌─── Null type: struct
│  │
EB 0B
```

10.9.3.1 Structs With Symbol Address Field Names

An opcode with a high nibble of `0xC_` indicates a struct with symbol address field names (which is similar to the only available encoding of structs in Ion 1.0. The lower nibble of the opcode indicates how many bytes were used to encode all of its nested `(field name, value)` pairs.

If the struct's encoded byte-length is too large to be encoded in a nibble, writers may use the `0xFC` opcode to write a variable-length struct with symbol address field names. The `0xFC` opcode is followed by a FlexUInt that indicates the byte length.

Each field in the struct is encoded as a FlexUInt representing the address of the field name's text in the symbol table, followed by an opcode-prefixed value.

Figure 79: Length-prefixed encoding of an empty struct ({})

```
┌──── An opcode in the range 0xC0-0xCF indicates a struct with symbol address field names
│┌─── A lower nibble of 0 indicates that the struct's fields took zero bytes to encode
C0
```

Figure 80: Length-prefixed encoding of {$10: 1, $11: 2}

```
┌──── An opcode in the range 0xC0-0xCF indicates a struct with symbol address field names
│  ┌─── Field name: FlexUInt 10 ($10)
│  │        ┌─── Field name: FlexUInt 11 ($11)
│  │        │
C6 15 51 01 17 51 02
      └─┬─┘    └─┬─┘
        1        2
```

Figure 81: Length-prefixed encoding of {$10: "variable length struct"}

```
┌──────────── Opcode `FC` indicates a variable length struct with symbol address field ↩
   names
│  ┌───────── Length: FlexUInt 25
│  │  ┌────── Field name: FlexUInt 10 ($10)
│  │  │  ┌─── Opcode `F8` indicates a variable length string
│  │  │  │  ┌ FlexUInt: 22 the string is 22 bytes long
│  │  │  │  │  v  a  r  i  a  b  l  e     l  e  n  g  t  h     s  t  r  u  c  t
FC 33 15 F8 2D 76 61 72 69 61 62 6c 65 20 6c 65 6e 67 74 68 20 73 74 72 75 63 74
               └─────────────────────────────┬─────────────────────────────┘
                                         UTF-8 bytes
```

10.9.3.2 Structs With `FlexSym` Field Names

Note

This encoding is very similar to structs with symbol address field names, but allows writers to choose between representing each field name as a symbol address (for example: `$10`) or as inline UTF-8 bytes (for example: `"foo"`). This encoding is potentially less dense, but offers writers significant flexibility over whether and when field names are added to the symbol table.

An opcode with a high nibble of `0xD_` indicates a struct with FlexSym field names. The lower nibble of the opcode indicates how many bytes were used to encode all of its nested `(field name, value)` pairs.

Warning

This form cannot be used to encode an empty struct; `0xD0` is a reserved opcode. Empty structs can be written using either the length-prefixed form `0xC0` or the delimited form `0xF3 0xF0`.

If the struct's encoded byte-length is too large to be encoded in a nibble, writers may use the `0xFD` opcode to write a variable-length struct with FlexSym field names. The `0xFD` opcode is followed by a FlexUInt that indicates the byte length.

Each field in the struct is encoded as a FlexSym field name, followed by an opcode-prefixed value.

Figure 82: Length-prefixed encoding of {"foo": 1, $11: 2}

```
┌──── Opcode with high nibble `D` indicates a struct with FlexSym field names
│ ┌── Length: 9
│ │  ┌─ FlexSym -3       ┌─ FlexSym: 11 ($11)
│ │  │  f  o  o          │
D9 FD 66 6F 6F 51 01 17 91 02
      └───┬───┘ └─┬┘    └┬┘
    3 UTF-8     1        2
     bytes
```

TODO: Demonstrate splicing macro values into the struct via FlexSym escape code `0x00`.

10.9.3.3 Delimited Structs

Opcode `0xF3` indicates the beginning of a delimited struct with FlexSym field names.

Unlike lists and S-expressions, structs cannot use opcode `0xF0` by itself to indicate the end of the delimited container. This is because `0xF0` is a valid `FlexSym` (a symbol with 16 bytes of inline text). To close the delimited struct, the writer emits a `0x00` byte (a `FlexSym` escape) followed by the opcode `0xF0`.

Note
While length-prefixed structs can choose between structs with symbol address field names and structs with FlexSym field names, delimited structs always use `FlexSym`-encoded field names.

Figure 83: Delimited encoding of the empty struct ({})

```
┌──── Opcode 0xF3 indicates the beginning of a delimited struct with `FlexSym` field names.
│  ┌──── FlexSym escape code 0x00: an opcode follows
│  │  ┌──── Opcode 0xF0 indicates the end of the most
│  │  │     recently opened delimited container
F3 00 F0
```

Figure 84: Delimited encoding of {"foo": 1, $11: 2}

```
┌──── Opcode 0xF3 indicates the beginning of a delimited struct with `FlexSym` field names.
│
│  ┌─ FlexSym -3      ┌─ FlexSym: 11 ($11)
│  │                  │          ┌──── FlexSym escape code 0x00: an opcode follows
│  │                  │          │  ┌──── Opcode 0xF0 indicates the end of the most
│  │   f  o  o        │          │  │     recently opened delimited container
F3 FD 66 6F 6F 51 01 17 91 02 00 F0
      └──┬───┘ └─┬┘    └─┬┘
      3 UTF-8    1       2
       bytes
```

10.10 Nulls

The opcode `0xEA` indicates an untyped null (that is: `null`, or its alias `null.null`).

The opcode `0xEB` indicates a typed null; a byte follows whose value represents an offset into the following table:

Byte	Type
`0x00`	`null.bool`
`0x01`	`null.int`
`0x02`	`null.float`
`0x03`	`null.decimal`
`0x04`	`null.timestamp`
`0x05`	`null.string`
`0x06`	`null.symbol`
`0x07`	`null.blob`
`0x08`	`null.clob`
`0x09`	`null.list`
`0x0A`	`null.sexp`
`0x0B`	`null.struct`

All other byte values are reserved for future use.

Note
Future versions of Ion may decide to generalize this into a "constants" table.

Figure 85: Encoding of null

```
┌──── The opcode `0xEA` represents a null (null.null)
EA
```


Figure 86: Encoding of null.string

```
┌──── The opcode `0xEB` indicates a typed null; a byte indicating the type follows
│  ┌──── Byte 0x05 indicates the type `string`
EB 05
```

10.11 Annotations

TODO: Decide whether we want an Ion 1.0-style double-length-prefixed sequence.

Annotations can be encoded either as symbol addresses or as FlexSyms. In both encodings, the annotations sequence appears just before the value that it decorates.

It is illegal for an annotations sequence to appear before any of the following:

- Another annotations sequence
- The end of the stream
- A NOP
- An E-expression (that is: a macro invocation). To add annotations to the expansion of an E-expression, see the `annotate` macro. (TODO: Link)

10.12 Annotations With Symbol Addresses

Opcodes `0xE4` through `0xE6` indicate one or more annotations encoded as symbol addresses. If the opcode is:

- `0xE4`, a single FlexUInt-encoded symbol address follows.
- `0xE5`, two FlexUInt-encoded symbol addresses follow.
- `0xE6`, a FlexUInt follows that represents the number of bytes needed to encode the annotations sequence, which can be made up of any number of `FlexUInt` symbol addresses.

Figure 87: Encoding of $10::false

```
┌──── The opcode `0xE4` indicates a single annotation encoded as a symbol address follows
│  ┌──── Annotation with symbol address: FlexUInt 10
E4 15 5F
      └── The annotated value: `false`
```

Figure 88: Encoding of $10::$11::false

```
┌──── The opcode `0xE5` indicates that two annotations encoded as symbol addresses follow
│  ┌──── Annotation with symbol address: FlexUInt 10 ($10)
│  │  ┌──── Annotation with symbol address: FlexUInt 11 ($11)
E5 15 17 5F
         └── The annotated value: `false`
```

Figure 89: Encoding of $10::$11::$12::false

```
┌──── The opcode `0xE6` indicates a variable-length sequence of symbol address annotations;
│     a FlexUInt follows representing the length of the sequence.
│   ┌──── Annotations sequence length: FlexUInt 3 with symbol address: FlexUInt 10 ($10)
│   │  ┌──── Annotation with symbol address: FlexUInt 10 ($10)
│   │  │  ┌──── Annotation with symbol address: FlexUInt 11 ($11)
│   │  │  │  ┌──── Annotation with symbol address: FlexUInt 12 ($12)
E5 07 15 17 19 5F
               └── The annotated value: `false`
```


10.13 Annotations With `FlexSym` Text

Opcodes `0xE7` through `0xE9` indicate one or more annotations encoded as FlexSyms.

If the opcode is:

- `0xE7`, a single `FlexSym`-encoded symbol follows.
- `0xE8`, two `FlexSym`-encoded symbols follow.
- `0xE9`, a `FlexUInt` follows that represents the byte length of the annotations sequence, which is made up of any number of annotations encoded as `FlexSyms`.

While this encoding is more flexible than annotations with symbol addresses, it can be slightly less compact when all the annotations are encoded as symbol addresses.

Figure 90: Encoding of $10::false

```
┌──── The opcode `0xE7` indicates a single annotation encoded as a FlexSym follows
│  ┌──── Annotation with symbol address: FlexSym 10 ($10)
E7 15 5F
      └── The annotated value: `false`
```

Figure 91: Encoding of foo::false

```
┌──── The opcode `0xE7` indicates a single annotation encoded as a FlexSym follows
│  ┌──── Annotation: FlexSym -3; 3 bytes of UTF-8 text follow
│  │   f  o  o
E7 FD 66 6F 6F 5F
      └──┬───┘ └── The annotated value: `false`
     3 UTF-8
      bytes
```

Note that `FlexSym` annotation sequences can switch between symbol address and inline text on a per-annotation basis.

Figure 92: Encoding of $10::foo::false

```
┌──── The opcode `0xE8` indicates two annotations encoded as FlexSyms follow
│  ┌──── Annotation: FlexSym 10 ($10)
│  │  ┌──── Annotation: FlexSym -3; 3 bytes of UTF-8 text follow
│  │  │   f  o  o
E8 15 FD 66 6F 6F 5F
         └──┬───┘ └── The annotated value: `false`
        3 UTF-8
         bytes
```

Figure 93: Encoding of $10::foo::$11::false

```
┌──── The opcode `0xE9` indicates a variable-length sequence of FlexSym-encoded annotations
│  ┌──── Length: FlexUInt 6
│  │  ┌──── Annotation: FlexSym 10 ($10)
│  │  │  ┌──── Annotation: FlexSym -3; 3 bytes of UTF-8 text follow
│  │  │  │            ┌──── Annotation: FlexSym 11 ($11)
│  │  │  │   f  o  o │
E9 0D 15 FD 66 6F 6F 17 5F
            └──┬───┘    └── The annotated value: `false`
           3 UTF-8
            bytes
```


10.14 NOPs

A NOP (short for "no-operation") is the binary equivalent of whitespace. NOP bytes have no meaning, but can be used as padding to achieve a desired alignment.

An opcode of 0xEC indicates a single-byte NOP pad. An opcode of 0xED indicates that a FlexUInt follows that represents the number of additional bytes to skip.

It is legal for a NOP to appear anywhere that a value can be encoded. It is not legal for a NOP to appear in annotation sequences or struct field names. If a NOP appears in place of a struct field *value*, then the associated field name is ignored; the NOP is immediately followed by the next field name, if any.

Figure 94: Encoding of a 1-byte NOP

```
┌──── The opcode `0xEC` represents a 1-byte NOP pad
│
EC
```

Figure 95: Encoding of a 4-byte NOP

```
┌──── The opcode `0xED` represents a variable-length NOP pad; a FlexUInt length follows
│  ┌───── Length: FlexUInt 2; two more bytes of NOP follow
│  │
ED 05 93 C6
      └─┬─┘
NOP bytes, values ignored
```

10.15 E-expression Arguments

The binary encoding of E-expressions (aka macro invocations) starts with the address of the macro to expand. The address can be encoded as part of the opcode or as a FlexUInt that follows the opcode.

The encoding of the E-expression's arguments depends on their respective types. Argument types can be classified as belonging to one of two categories: tagged encodings and tagless encodings.

10.15.1 Tagged Encodings

Tagged types are argument types whose encoding begins with an opcode, sometimes informally called a 'tag'. These include the core types and the abstract types.

10.15.1.1 Core types

The *core types* are the 13 types in the Ion data model:

null | bool | int | float | decimal | timestamp | string | symbol | blob | clob | list | sexp | struct

10.15.1.2 Abstract types

The *abstract types* are unions of two or more of the core types.

Abstract type	Included Ion types
any	All core Ion types
number	int, float, decimal
exact	int, decimal
text	string, symbol

Abstract type	Included Ion types
lob	blob, clob
sequence	list, sexp

10.15.1.3 Tagged E-expression Argument Encoding

When a macro parameter has a tagged type, the encoding of that parameter's corresponding argument in an E-expression is identical to how it would be encoded anywhere else in an Ion stream: it has a leading opcode that dictates how many bytes follow and how they should be interpreted. This is very flexible, but makes it possible for writers to encode values that conflict with the parameter's declared type. Because of this, the macro expander will read the argument and then check its type against the parameter's declared type. If it does not match, the macro expander must raise an error.

Macro `foo` (defined below) is used in this section's subsequent examples to demonstrate the encoding of tagged-type arguments.

Figure 96: Definition of example macro foo at address 0

```
(macro
    foo             // Macro name
    [(x number!)] // Parameters
    /*...*/         // Template (elided)
)
```

Figure 97: Encoding of E-expression (:foo 3.14e)

```
┌──── The opcode is less than 0x40, so it is an E-expression invoking the macro at
│     address 0: `foo`. `foo` takes a tagged number as a parameter (`x`), so an opcode  ↩
    follows.
│  ┌──── Opcode 0x5B indicates a 2-byte float; an IEEE-754 half-precision float follows
│  │
00 5B 42 47
      └─┬─┘
      3.14e0

// The macro expander confirms that `3.14e0` (a `float`) matches the expected type: `number ↩
    `.
```

Figure 98: Encoding of E-expression (:foo 9)

```
┌──── The opcode is less than 0x40, so it is an E-expression invoking the macro at
│     address 0: `foo`. `foo` takes a tagged number as a parameter (`x`), so an opcode  ↩
    follows.
│  ┌──── Opcode 0x51 indicates a 1-byte integer. A 1-byte FixedInt follows.
│  │  ┌──── A 1-byte FixedInt: 9
00 51 09

// The macro expander confirms that `9` (an `int`) matches the expected type: `number`.
```

Figure 99: Encoding of E-expression (:foo $10::9)

```
┌──── The opcode is less than 0x40, so it is an E-expression invoking the macro at
│     address 0: `foo`. `foo` takes a tagged number as a parameter (`x`), so an opcode  ↩
    follows.
│  ┌──── Opcode 0xE4 indicates a single annotation with symbol address. A FlexUInt follows.
│  │  ┌──── Symbol address: FlexUInt 10 ($10); an opcode for the annotated value follows.
│  │  │  ┌──── Opcode 0x51 indicates a 1-byte integer
│  │  │  │  ┌──── 1-byte FixedInt 9
00 E4 15 51 09

// The macro expander confirms that `$10::9` (an annotated `int`) matches the expected type ↩
    : `number`.
```


Figure 100: Encoding of E-expression (:foo null.int)

```
┌──── The opcode is less than 0x40, so it is an E-expression invoking the macro at
│     address 0: `foo`. `foo` takes a tagged number as a parameter (`x`), so an opcode ↩
   follows.
│  ┌──── Opcode 0xEB indicates a typed null. A 1-byte FixedUInt follows indicating the type ↩
   .
│  │  ┌──── Null type: FixedUInt: 1; integer
00 EB 01

// The macro expander confirms that `null.int` matches the expected type: `number`.
```

Figure 101: Encoding of E-expression (:foo null)

```
┌──── The opcode is less than 0x40, so it is an E-expression invoking the macro at
│     address 0: `foo`. `foo` takes a tagged number as a parameter (`x`), so an opcode ↩
   follows.
│  ┌──── Opcode 0xEA represents an untyped null (aka `null.null`)
00 EA

// The macro expander confirms that `null` matches the expected type: `number`
```

Figure 102: Encoding of E-expression (:foo (:bar))

```
// A second macro definition at address 1
(macro
    bar // Macro name
    ()  // Parameters
    5   // Template; invocations of `bar` always expand to `5`.
)

┌──── The opcode is less than 0x40, so it is an E-expression invoking the macro at
│     address 0: `foo`. `foo` takes a tagged int as a parameter (`x`), so an opcode follows ↩
   .
│  ┌──── Opcode 0x01 is less than 0x40, so it is an E-expression invoking the macro
│  │     at address 1: `bar`. `bar` takes no parameters, so no bytes follow.
00 01

// The macro expander confirms that the expansion of `(:bar)` (that is: `5`) matches
// the expected type: `number`.
```

Figure 103: Encoding of illegal E-expression (:foo "hello")

```
┌──── The opcode is less than 0x40, so it is an E-expression invoking the macro at
│     address 0, `foo`. `foo` takes a tagged int as a parameter (`x`), so an opcode follows ↩
   .
│  ┌──── Opcode 0x85 indicates a 5-byte string. 5 UTF-8 bytes follow.
│  │  h  e  l  l  o
00 85 68 65 6C 6C 6F
      └──────┬──────┘
        UTF-8 bytes

// ERROR: Expected a `number` for `foo` parameter `x`, but found `string`
```

10.15.2 Tagless Encodings

In contrast to tagged encodings, *tagless encodings* do not begin with an opcode. This means that they are potentially more compact than a tagged type, but are also less flexible. Because tagless encodings do not have an opcode, they cannot represent E-expressions, annotation sequences, or `null` values of any kind.

Tagless types include the primitive types and macro shapes.

10.15.2.1 Primitive Types

Primitive types are self-delineating, either by having a statically known size in bytes or by including length information in their encoding.

Primitive types include:

Ion type	Primitive type	Size in bytes	Encoding
int	uint8	1	FixedUInt
	uint16	2	
	uint32	4	
	uint64	8	
	compact_uint	variable	FlexUInt
	int8	1	FixedInt
	int16	2	
	int32	4	
	int64	8	
	compact_int	variable	FlexInt
float	float16	2	IEEE-754 half-precision floating point format
	float32	4	IEEE-754 single-precision floating point format
	float64	8	IEEE-754 double-precision floating point format
symbol	compact_symbol	variable	FlexSym

TODO:

- Finalize names for primitive types. (`compact_`? `plain_`?)
- Do we need a `compact_string` encoding? It saves a byte for string lengths >16 and <128.
- Do we need other int sizes? `int24`? `int40`?

10.15.2.2 Macro Shapes

The term *macro shape* describes a macro that is being used as the encoding of an E-expression argument. They are considered "shapes" rather than types because while their encoding is always statically known, the types of data produced by their expansion is not. A single macro can produce streams of varying length and containing values of different Ion types depending on the arguments provided in the invocation.

See the Macro Shapes section of *Macros by Example* for more information.

10.16 Encoding E-expressions With Multiple Arguments

E-expression arguments corresponding to each parameter are encoded one after the other moving from left to right.

Figure 104: Definition of macro foo at address 0

```
(macro foo              // Macro name
  [                     // Parameters
    (a string!),
    (b compact_symbol!),
    (c uint16!)
  ]
  /* ... */             // Body (elided)
)
```

Figure 105: Encoding of E-expression for macro with multiple parameters: (:0 "hello" baz 512)


```
┌──── The opcode is less than 0x40, so it is an E-expression invoking the macro at
│     address 0, `foo`. `foo'`s first parameter is a string, so an opcode follows.
│
│  ┌──── Opcode 0x85 indicates a 5-byte string. 5 UTF-8 bytes follow.
│  │
│  │                    ┌──── `foo'`s second parameter is a compact_symbol, so a `FlexSym`  ↩
   follows.
│  │                    │    FlexSym -3: 3 bytes of UTF-8 text follow.
│  │                    │
│  │                    │          ┌──── `foo'`s third parameter is a uint16, so a 2-byte
│  │                    │          │     2-byte `FixedUInt` follows.
│  │                    │          │     FixedUInt: 512
│  │  h  e  l  l  o     │  b  a  z │
00 85 68 65 6C 6C 6F FD 62 61 7A 00 20
      └──────┬──────┘    └───┬───┘
        UTF-8 bytes     UTF-8 bytes
```

10.17 Argument Encoding Bitmap (AEB)

The examples in previous sections have only shown how to encode invocations of macros which have either no parameters at all (aka *constants*) or whose parameters all have a cardinality of exactly-one.

If a macro has any parameters with a cardinality of zero-or-one (`?`), zero-or-more (`*`), or one-or-more (`+`), then E-expressions invoking that macro will begin with an *argument encoding bitmap* (AEB). An AEB is a series of bits that correspond to a macro parameter and communicate additional information about how the arguments corresponding to that parameter have been encoded in the current E-expression. In particular, the AEB indicates whether a parameter that accepts `(:void)` has any arguments at all, and how a grouped parameter's arguments have been delimited.

The number of bits allotted to each parameter is determined by its cardinality, as shown in the table below; each parameter can have 0, 1, or 2 bits.

Grouping Mode	Cardinality	Example parameter signature	Number of bits	Bit(s) value	Encoding
Ungrouped	Exactly-one	`(x int!)`	0	*n/a*	One expression
	Zero-or-one	`(x int?)`	1	`0`	No expression; equivalent to `(:void)`
				`1`	One expression
	Zero-or-more	`(x int*)`		`0`	No expression; equivalent to `(:void)`
				`1`	One expression
	One-or-more	`(x int+)`	0	*n/a*	One expression
Grouped	Zero-or-more	`(x [int])` `(x int...)`	2	`00`	No expression; equivalent to `(:void)`
				`01`	One expression
				`10`	Length-prefixed expression group
				`11`	Delimited expression group
	One-or-more	`(x )` + `(x int\...)`		`00`	*Illegal.* One-or-more forbids `(:void)`.
				`01`	One expression
				`10`	Length-prefixed expression group
				`11`	Delimited expression group

The total number of bits in the AEB can be calculated by analyzing the signature of the macro being invoked. If the macro has no parameters or all of its parameters have a cardinality of either exactly-one or one-or-more, no bits are required; the AEB will be omitted altogether. If the macro has many parameters with a cardinality other than exactly-one, it is possible for the AEB to require more than one byte to encode; in such cases, the bytes are written in little-endian order. AEB bytes can contain unused bits.

Bits are assigned to the parameters in a macro's signature from left to right. Bits are assigned from least significant to most significant (commonly: right-to-left).

Example parameter sequence	Bit assignments	Total bits
`()`	*No AEB*	0
`((a int!) (b string!) (c float!))`	*No AEB*	0
`((a int!) (b string!) (c float?))`	`-------c`	1
`((a int!) (b string?) (c float!))`	`-------b`	1
`((a int!) (b string*) (c float?))`	`------cb`	2
`((a int*) (b string!) (c [float]))`	`-----cca`	3
`((a int*) (b [string]) (c [float]))`	`---ccbba`	5
`((a [int]) (b [string]) (c [float]+))`	`--ccbbaa`	6
`((a int*) (b [string]) (c [float]) (d [bool]) (e blob...))`	`eddccbba -------e`	9

10.18 Expression Groups

Grouped parameters can be encoded using either a length-prefixed or delimited expression group encoding.

The example encodings in the following sections refer to this macro definition:

Figure 106: Definition of macro foo at address 0

```
(macro
    foo           // Macro name
    [(x [int])]   // Parameters; 'x' is a grouped parameter
    /*...*/       // Body (elided)
)
```

10.18.1 Length-prefixed Expression Groups

If a grouped parameter's AEB bits are `0b10`, then the argument expressions belonging to that parameter will be prefixed by a `FlexUInt` indicating the number of bytes used to encode them.

Figure 107: Length-prefixed encoding of (:foo [1, 2, 3])

```
┌──── The opcode is less than 0x40, so it is an E-expression invoking the macro at
│     address 0: 'foo'. 'foo' takes a group of int expressions as a parameter ('x'),
│     so an argument encoding bitmap (AEB) follows.
│  ┌──── AEB: 0b0000_0010; the arguments for grouped parameter 'x' have been encoded
│  │     as a length-prefixed expression group. A FlexUInt length prefix follows.
│  │  ┌──── FlexUInt: 6; the next 6 bytes are an 'int' expression group.
│  │  │
00 02 0D 51 01 51 02 51 03
         └─┬─┘ └─┬─┘ └─┬─┘
           1     2     3
```

10.18.2 Delimited Expression Groups

If a grouped parameter's AEB bits are `0b11`, then the argument expressions belonging to that parameter will be encoded in a delimited sequence. Delimited sequences are encoded differently for tagged types and tagless types.

10.18.2.1 Delimited Tagged Expression Groups

Tagged type encodings begin with an opcode; a delimited sequence of tagged arguments is terminated by the closing delimiter opcode, `0xF0`.

Figure 108: Delimited encoding of (:foo [1, 2, 3])

```
┌──── The opcode is less than 0x40, so it is an E-expression invoking the macro at
│     address 0: `foo`. `foo` takes a group of int expressions as a parameter (`x`),
│     so an argument encoding bitmap (AEB) follows.
│  ┌──── AEB: 0b0000_0011; the arguments for grouped parameter `x` have been encoded
│  │     as a delimited expression group. A series of tagged `int` expressions follow.
│  │                      ┌──── Opcode 0xF0 ends the expression group.
│  │                      │
00 03 51 01 51 02 51 03 F0
      └─┬─┘ └─┬─┘ └─┬─┘
        1     2     3
```

10.18.2.2 Delimited Tagless Expression Groups

Tagless type encodings do not have an opcode, and so cannot use the closing delimiter opcode--`0xF0` is a valid first byte for many tagless encodings.

Instead, tagless expressions are grouped into 'pages', each of which is prefixed by a FlexUInt representing a count (not a byte-length) of the expressions that follow. If a prefix has a count of zero, that marks the end of the sequence of pages.

Figure 109: Definition of macro compact_foo at address 1

```
(macro
    compact_foo           // Macro name
    [(x [compact_int])]   // Parameters; `x` is a grouped parameter
    /*...*/               // Body (elided)
)
```

Figure 110: Delimited encoding of (:compact_foo [1, 2, 3]) using a single page

```
┌──── The opcode is less than 0x40, so it is an E-expression invoking the macro at
│     address 0: `foo`. `foo` takes a group of int expressions as a parameter (`x`),
│     so an argument encoding bitmap (AEB) follows.
│  ┌──── AEB: 0b0000_0011; the arguments for grouped parameter `x` have been encoded
│  │     as a delimited expression group. Count-prefixed pages of `compact_int`
│  │     expressions follow.
│  │   ┌──── Count prefix: FlexUInt 3; 3 `compact_int`s follow.
│  │   │           ┌──── Count prefix: FlexUInt 0; no more pages follow.
│  │   │           │
00 03 07 03 05 07 01
         └───┬───┘
         First page: 1, 2, 3
```

Figure 111: Delimited encoding of (:compact_foo [1, 2, 3]) using two pages

```
┌──── The opcode is less than 0x40, so it is an E-expression invoking the macro at
│     address 0: `foo`. `foo` takes a group of int expressions as a parameter (`x`),
│     so an argument encoding bitmap (AEB) follows.
│  ┌──── AEB: 0b0000_0011; the arguments for grouped parameter `x` have been encoded
│  │     as a delimited expression group. Count-prefixed pages of `compact_int`
│  │     expressions follow.
│  │   ┌──── Count prefix: FlexUInt 2; 2 `compact_int`s follow.
│  │   │         ┌──── Count prefix: FlexUInt 1; a single `compact_int` follows.
│  │   │         │   ┌──── Count prefix: FlexUInt 0; no more pages follow.
│  │   │         │   │
00 03 05 03 05 03 07 01
         └─┬─┘    └─ Second page: 3
           │
         First page: 1, 2
```

Chapter 11

Domain Grammar

This chapter presents Ion 1.1's *domain grammar*, by which we mean the grammar of the domain of values that drive Ion's encoding features.

We use a BNF-like notation for describing various syntactic parts of a document, including Ion data structures. In such cases, the BNF should be interpreted loosely to accommodate Ion-isms like commas and unconstrained ordering of struct fields.

All `()` `[]` `{}` below are literal tokens of Ion syntax. Single-quoted **`'?'`** and **`'*'`** denote literal Ion symbols, while unquoted |, ?, and * are BNF notation.

11.1 Documents

TODO this section needs much work.

document	::=	*segment**
segment	::=	*ivm*? *value** *directive*?
directive	::=	*symtab-directive* | *encoding-directive*

11.2 Encoding Directives

encoding-directive	::=	`$ion_encoding`**`::`** `(` *retention*? *module-decl** *symtab*? *top-mactab*? `)`
retention	::=	`(` **`retain`** *retainees* `)`
retainees	::=	**`'*'`** | *module-name**
module-decl	::=	*dependency* | *inline-module-def*
dependency	::=	*load-decl* | *use-decl* | *import-decl*
use-decl	::=	`(` **`use`** *use-item** `)`
use-item	::=	*module-name* | *load-decl*
symtab	::=	`(` **`symbol_table`** *symtab-item** `)`
symtab-item	::=	*module-name* | `[` *text** `]`
top-mactab	::=	`(` **`macro_table`** *module-name** `)`

11.2.1 Catalog Access

load-decl	::=	`(` **`load`** *load-body* `)`
import-decl	::=	`(` **`import`** *load-body* `)`
load-body	::=	*module-name catalog-name catalog-version symbol-maxid*?
catalog-name	::=	*unannotated-string*

catalog-version	::=	*unannotated-uint*
symbol-maxid	::=	*unannotated-uint*

11.3 Macro References

macro-ref	::=	*macro-name* | *local-ref* | *qualified-ref*
local-ref	::=	⟨symbol of the form `':`*name-or-address*`'`⟩
qualified-ref	::=	⟨symbol of the form `':`*module-name*`:`*name-or-address*`'`⟩
module-name	::=	*unannotated-identifier-symbol*
macro-name	::=	*unannotated-identifier-symbol*
macro-address	::=	*unannotated-uint*
name-or-address	::=	*macro-name* | *macro-address*

11.4 Module Definitions

inline-module-def	::=	`(` **`module`** *module-name module-body* `)`
shared-module-def	::=	`$ion_shared_module::`*ion-version-marker*`::(` *catalog-key module-body* `)`
catalog-key	::=	`(` **`catalog_key`** *catalog-name catalog-version* `)`

11.4.1 Module Bodies

module-body	::=	*dependency* symtab? macro-alias* module-mactab?*
macro-alias	::=	`(` **`alias`** *macro-name macro-ref* `)`
module-mactab	::=	`(` **`macro_table`** *macro-or-export** `)`
macro-or-export	::=	*macro-defn* | *export*
export	::=	`(` **`export`** *export-item** `)` | *module-name*
export-item	::=	*macro-ref* | `(` **`from`** *module-name name-or-address** `)`

11.5 Macro Definitions

macro-defn	::=	`(` **`macro`** *macro-name? signature template* `)`
signature	::=	*param-specs result-spec?*
param-specs	::=	`(` *param-spec* rest-spec?* `)` | `[` *param-spec* rest-spec?* `]`
param-spec	::=	*param-name* | `(` *param-name param-shape* `)`
rest-spec	::=	`(` *param-name rest-shape* `)`
param-name	::=	*unannotated-identifier-symbol*
param-shape	::=	*simple-shape* | *grouped-shape*
simple-shape	::=	*tagged-type? tagged-cardinality?* | *tagless-type tagless-cardinality?*
tagged-cardinality	::=	**`!`** | **`+`** | **`'?'`** | **`'*'`**
tagless-cardinality	::=	**`'?'`**
grouped-shape	::=	`[` *any-type?* `]` *grouped-cardinality?*
grouped-cardinality	::=	**`'+'`**
rest-shape	::=	*any-type? rest-cardinality*
rest-cardinality	::=	**`...`** | **`...+`**
any-type	::=	*tagged-type* | *tagless-type*
tagged-type	::=	*abstract-type* | *concrete-type*
tagless-type	::=	*primitive-type* | *macro-ref*
abstract-type	::=	**`any`** | **`number`** | **`exact`** | **`text`** | **`lob`** | **`sequence`**

concrete-type	::=	**'null'** | **bool** | **timestamp** | **int** | **decimal** | **float** | **string** | **symbol** | **blob** | **clob** | **list** | **sexp** | **struct**
primitive-type	::=	**var_symbol** | **var_string** | **var_int** | **var_uint** | **uint8** | **uint16** | **uint32** | **uint64** | **int8** | **int16** | **int32** | **int64** | **float16** | **float32** | **float64**
result-spec	::=	**->** *tagged-type tagged-cardinality*

11.6 Template Expressions

template	::=	*identifier* | *literal* | *quasi-literal* | *special-form* | *macro-invocation*
literal	::=	*null* | *bool* | *int* | *float* | *decimal* | *timestamp* | *string* | *blob* | *clob*
quasi-literal	::=	[*template**] | { *quasi-field** }
quasi-field	::=	*text* **:** *template*
special-form	::=	(**literal** *datum*) | (**if_void** $template_{cond}$ $template_{then}$ $template_{else}$) | (**if_single** $template_{cond}$ $template_{then}$ $template_{else}$) | (**if_many** $template_{cond}$ $template_{then}$ $template_{else}$) | (**for** [*for-clause**] $template_{body}$)
for-clause	::=	(*identifier* $template_{in}$)
macro-invocation	::=	(*macro-ref macro-arg**)
macro-arg	::=	*template* | [*template**] // *Very* roughly

Important
Special forms take precedence over macro invocations. Use a *local-ref* or *qualified-ref* to invoke a macro whose name shadows a special-form keyword.

Important
The syntax of *macro-arg*s is constrained by the macro expander, based on the signature of the invoked macro.

11.7 Backwards Compatibility

11.7.1 Symbol Table Directives

symtab-directive	::=	TODO

11.7.2 Tunneled Modules

shared-symtab	::=	`$ion_shared_symbol_table::{` `name` **:** *catalog-name* `version` **:** *catalog-version* `symbols` **:** [string*] `module` **:** *tunneled-module-def* }
tunneled-module-def	::=	*ion-version-marker* **::** (*tunneled-module-body*)
tunneled-module-body	::=	*dependency* macro-alias* module-mactab*

Tip
A tunneled module may not have a `symbol_table` clause; symbols must be defined in the legacy `symbols` field.

Chapter 12

Glossary

actual arity

The number of subforms (arguments or argument groups) in a macro invocation. A macro can be *fixed arity* or *variable arity*.

argument

A single sub-expression within a macro invocation, corresponding to one of the macro's parameters. This is by default a one-to-one relation, but a parameter's grouping mode can change this to a many-to-one relation.

argument group

The concrete syntax for a *grouped parameter*. In a macro invocation, a list containing multiple arguments, delimited explicitly with `[...]` notation in Ion text. Used to express parameters that accept more than one argument.

cardinality

Describes the number of values that a parameter will accept when the macro is invoked. One of zero-or-one, exactly-one, zero-or-more, or one-or-more. Specified in a signature by one of the modifiers **?**, **!**, *****, ***`, `*\...*` or `*\....**

declaration

The association of a name with an entity (for example, a module or macro). See also *definition*. Not all declarations are definitions: some introduce new names for existing entities.

definition

The specification of a new entity.

directive

A keyword or unit of data in an Ion document that affects the encoding environment, and thus the way the document's data is decoded. In Ion 1.0 there are two directives: *Ion version markers*, and the *symbol table directives*. Ion 1.1 adds *encoding directives*.

document

A stream of octets conforming to either the Ion text or binary specification. Can consist of multiple *segments*, perhaps using varying versions of the Ion specification. A document doesn't necessarily exist as a file, and isn't necessarily finite.

E-expression

See *encoding expression*.

encoding directive

In an Ion 1.1 segment, a top-level struct annotated with `$ion_encoding`. Defines a new encoding environment for the segment immediately following it. The *symbol table directive* is effectively a less capable alternative syntax.

encoding environment

The context-specific data maintained by an Ion implementation while encoding or decoding data. In Ion 1.0 this consists of the current symbol table; in Ion 1.1 this is expanded to also include the Ion spec version, the current macro table, and a collection of available modules.

encoding expression

The invocation of a macro in encoded data, aka E-expression. Starts with a macro reference denoting the function to invoke. The Ion text format uses “smile syntax” `(:macro ...)` to denote E-expressions. Ion binary devotes a large number of opcodes to E-expressions, so they can be compact.

fixed arity

Describes a macro without optional or rest parameters, so invocations must have actual arity that equals the macro’s formal arity.

formal arity

The number of parameters declared by a macro. Due to *optional parameters* and *rest parameters*, the *actual arity* of a macro invocation may differ from its formal arity.

grouped parameter

A macro parameter that accepts multiple arguments in an *argument group* when the macro is invoked. See also *grouping mode*.

grouping mode

One of three ways that a macro parameter is given arguments. A *simple parameter* accepts one argument, a *grouped parameter* accepts a list of arguments, and a *rest parameter* accepts “all the rest” of the trailing arguments without a grouping list.

Ion version marker

A keyword directive that denotes the start of a new segment encoded with a specific Ion version. Also known as “IVM”.

macro

A transformation function that accepts some number of streams of values, and produces a stream of values.

macro definition

Specifies a macro in terms of a *signature* and a *template*.

macro reference

Identifies a macro for invocation, alias, or exporting. Must always be unambiguous. Lexically scoped, and never a “forward reference” to a macro that’s declared later in the document.

module

The data entity that defines and exports both symbols and macros. Modules are imported by encoding directives then installed into the local symbol and/or macro tables.

optional parameter

A parameter that can have its corresponding subform(s) omitted when the macro is invoked. A parameter is optional if it is *voidable* and all following arguments are also voidable.

parameter

A named input to a macro, as defined by its signature. At expansion time a parameter produces a stream of values.

qualified macro reference

A macro reference that consists of a module name and either a macro name exported by that module, or a numeric address within the range of the module’s exported macro table. In text, these look like `:module-name:name-or-address`.

quasi-literal

A template, denoted as a list or struct, that is *partly* ("quasi-") literal. List-shaped templates treat the elements as nested templates. Struct-shaped templates treat the field names as literal, but the corresponding values as templates. S-expressions denote operator invocations and are not treated quasi-literally.

rest parameter

A macro parameter—always the final parameter—declared with the `...` or `...+` modifier, that accepts all remaining arguments to the macro as if they were in an implicit *argument group*. Similar to "varargs" parameters in Java and other languages. See also *grouping mode*.

segment

A contiguous partition of a document that uses the same encoding environment. Segment boundaries are caused by directives: an IVM starts a new segment, while `$ion_symbol_table` and `$ion_encoding` directives end segments (with a new one starting immediately afterwards).

signature

The part of a macro definition that specifies its "calling convention", in terms of the shape, type, and cardinality of arguments it accepts, and the type and cardinality of the results it produces.

simple parameter

A macro parameter that matches a single argument when the macro is invoked. See also *grouping mode*.

subform

A nested portion within some syntactic form of the module or macro declarations.

symbol table directive

A top-level struct annotated with `$ion_symbol_table`. Defines a new encoding environment without any macros. Valid in Ion 1.0 and 1.1.

system symbol

A symbol provided by the Ion implementation via the system module `$ion`. System symbols are available at all points within an Ion document, though the selection of symbols varies by segment according to its Ion version.

system macro

A macro provided by the Ion implementation via the system module `$ion`. System macros are available at all points within Ion 1.1 segments.

system module

A standard module named `$ion` that is provided by the Ion implementation, implicitly installed so that the system symbols and system macros are available at all points within a document. Subsumes the functionality of the Ion 1.0 system symbol table.

template

The part of a macro definition that expresses its transformation of inputs to results.

unqualified macro reference

A macro reference that consists of either a macro name or numeric address, without a qualifying module name. These are resolved using lexical scope and must always be unambiguous.

variable arity

Describes a macro with optional and/or rest parameters, so invocations may have actual arity different from the macro's formal arity.

void

An empty stream of values. Produced by the system macro `void` as in the E-expression `(:void)`.

voidable

Describes a parameter that accepts void, aka the empty stream. Such parameters have cardinality zero-or-one or zero-or-more.

