

# MASR — Meta Abstract Semantics Representation

Brian Beckman

10 Apr 2023

## Contents

<b>1</b>	<b>Abstract</b>	<b>3</b>
<b>2</b>	<b>Summary and Cheat Sheet</b>	<b>4</b>
2.1	Tables . . . . .	4
2.2	Multi-Specs . . . . .	6
2.3	Nested Multi-Specs . . . . .	7
2.4	Light Sugar, Heavy Sugar . . . . .	8
<b>3</b>	<b>Issues with ASDL</b>	<b>11</b>
3.1	ASDL is not Type-Checked . . . . .	11
3.2	ASDL is Moribund . . . . .	11
3.3	ASDL is Incomplete . . . . .	11
3.4	ASDL's ASR is Volatile . . . . .	11
3.5	ASDL is Ambiguous . . . . .	11
<b>4</b>	<b>Clojure Solves ASDL Issues</b>	<b>12</b>
4.1	Clojure Is a Complete Fit for ASR . . . . .	12
4.2	Clojure is not Moribund . . . . .	12
4.3	Clojure Has Type-Checking Tools . . . . .	12
4.4	ASR in Clojure Solves Volatility . . . . .	12
<b>5</b>	<b>MASR Definitions</b>	<b>13</b>
<b>6</b>	<b>MASR Tenets</b>	<b>15</b>
<b>7</b>	<b>Base Specs</b>	<b>16</b>
7.1	Atoms: <code>int</code> , <code>float</code> , <code>bool</code> , <code>nat</code> . . . . .	16
7.2	Notes . . . . .	17
<b>8</b>	<b>Term-Like Nodes</b>	<b>18</b>
8.1	dimensions [ <i>sic</i> ] . . . . .	18
8.2	identifier [ <i>sic</i> ] . . . . .	20
8.3	identifiers [ <i>sic</i> ] . . . . .	22

<b>9</b>	<b>Specs</b>	<b>25</b>
9.1	intent . . . . .	26
9.2	<b>TODO</b> unit . . . . .	32
9.3	<b>TODO</b> symbol . . . . .	33
9.4	storage_type . . . . .	34
9.5	access . . . . .	35
9.6	<b>TODO</b> deftype . . . . .	35
9.7	presence . . . . .	36
9.8	abi . . . . .	37
9.9	<b>TODO</b> stmt . . . . .	40
9.10	<b>TODO</b> expr . . . . .	41
9.11	ttype . . . . .	42
9.12	<b>TODO</b> restriction_arg . . . . .	42
9.13	<b>TODO</b> binop . . . . .	43
9.14	<b>TODO</b> logicalbinop . . . . .	44
9.15	<b>TODO</b> cmpop . . . . .	45
9.16	<b>TODO</b> integerboz . . . . .	46
9.17	<b>TODO</b> arraybound . . . . .	47
9.18	<b>TODO</b> arraystorage . . . . .	48
9.19	<b>TODO</b> cast_kind . . . . .	49
9.20	dimension . . . . .	50
9.21	<b>TODO</b> alloc_arg . . . . .	52
9.22	<b>TODO</b> attribute . . . . .	53
9.23	<b>TODO</b> attribute_arg . . . . .	54
9.24	<b>TODO</b> call_arg . . . . .	55
9.25	<b>TODO</b> tbind . . . . .	56
9.26	<b>TODO</b> array_index . . . . .	57
9.27	<b>TODO</b> do_loop_head . . . . .	58
9.28	<b>TODO</b> case_stmt . . . . .	59
9.29	<b>TODO</b> type_stmt . . . . .	60
9.30	<b>TODO</b> enumtype . . . . .	61
<b>10</b>	<b>Implicit Terms</b>	<b>62</b>
10.1	syntab_id . . . . .	62
10.2	<b>TODO</b> symbol_table . . . . .	63

# 1 Abstract

Abstract Semantics Representation (ASR) is a novel intermediate representation (IR)<sup>1</sup> for a new collection of LCompilers *[sic]*.<sup>2</sup> ASR is independent of the particular programming language under compilation. Current compiler front-ends targeting ASR include LFortran<sup>3</sup> and LPython.<sup>4</sup> ASR is also agnostic to the compiler back end. Current back ends targeted *from* ASR include LLVM, x86, C, and WASM<sup>5</sup>

Being agnostic means that it is easy to write new compilers, both at the front end and the back end. LCompiler back ends are reusable because ASR eliminates all original language syntax from the IR. For example. LFortran predates LPython. When the need for a Python compiler arose, only a front end was necessary. It took only a few days to create a new end-to-end compiler, LPython.

In addition to being more flexible, LCompilers are faster than average because optimizers are not hampered by impertinent syntactic structure.

Current specifications for ASR are written in ASDL,<sup>6</sup> a metalanguage similar in spirit to yacc but less rich, by design.<sup>7</sup> To build an LCompiler like LFortran or LPython, the ASDL grammar<sup>8</sup> for ASR is parsed, and a library, libasr,<sup>9</sup> in C++, is generated. Compiler front ends call functions in this library to manipulate ASR and to send it to the back ends.

ASDL has several deficiencies, and MASR,<sup>10</sup> described in this document, alleviates them. Chief among the deficiencies is the lack of type-checking. MASR adds a meta-type system to ASR.

We aim to replace ASDL with MASR and to integrate MASR with the LCompiler code base. When integrated in the future, MASR will be called LASR.

This document is pedagogical, both explaining MASR and teaching how to extend and maintain its Clojure code.

This document may lag the Clojure code. It may also lag the current state of libasr, at least until MASR replaces ASDL. The document mirrors an ASDL snapshot.<sup>8</sup>

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Intermediate\\_representation](https://en.wikipedia.org/wiki/Intermediate_representation)

<sup>2</sup><https://github.com/lcompilers/libasr>

<sup>3</sup><https://lfortran.org/>

<sup>4</sup><https://lpython.org/>

<sup>5</sup><https://webassembly.org/>

<sup>6</sup>[https://en.wikipedia.org/wiki/Abstract-Type\\_and\\_Scheme-Definition\\_Language](https://en.wikipedia.org/wiki/Abstract-Type_and_Scheme-Definition_Language)

Language

<sup>7</sup><https://en.wikipedia.org/wiki/Yacc>

<sup>8</sup>[https://github.com/rebcabin/masr/blob/main/ASR\\_2023\\_APR\\_06\\_snapshot.asdl](https://github.com/rebcabin/masr/blob/main/ASR_2023_APR_06_snapshot.asdl)

<sup>9</sup><https://github.com/lfortran/lfortran/tree/c648a8d824242b676512a038bf2257f3b28dad3b/src/libasr>

<sup>10</sup>pronounced “maser;” it is a Physics pun

## 2 Summary and Cheat Sheet

### 2.1 Tables

MASR defines *terms* and *heads* for the semantics of programs. Terms are top-most in the ASR grammar and heads are at the bottom level. Terms and heads have *specs*. Specs double as *types*, constituting ASR's meta-type system.

The following tables summarize this document via conforming examples. The examples are usually written in short, heavily sugared form (see Section 2.4 for light versus heavy sugar). Every spec *qua* type has a full form as well as one or more sugared forms. Sugared forms are shorter and recommended much of the time.

Equally important are non-conforming examples. See the body of this document, the tests in `specs.clj`, and the `deftest`'s in `core_tests` for many non-conforming examples.

Specs are registered in Clojure via *fully qualified keywords*.<sup>11</sup> All MASR spec keywords are in namespace `masr.specs`. For example, `::nat` is short for `:masr.specs/nat`.

Table 1: Atomic and Naked Specs: No Suga)

Spec	Predicate	link	Example
<code>::syntab-id</code>	<code>::nat</code>	10.1	42
<code>::bool</code>	<code>boolean?</code>	7.1	<code>true</code>
<code>::float</code>	<code>float?</code>	7.1	3.142
<code>::int</code>	<code>int?</code>	7.1	-1789

Table 2: Top-Level *term-like* Specs

Spec	link	Sugared Conforming Example
<code>::nat</code>	7.1.1	<code>(nat 42)</code>
<code>::identifier</code>	8.2	<code>(identifier 'boofar)</code>
<code>::identifier-set</code>	8.3	<code>(identifier-set ['a 'a])</code>
<code>::identifier-list</code>	8.3	<code>(identifier-list ['a 'a])</code>
<code>::identifier-suit</code>	8.3	<code>(identifier-suit ['a 'b])</code>
<code>::dimensions</code>	8.1	<code>(dimensions [[6 60] [1 42]])</code>

<sup>11</sup><https://clojure.org/guides/spec>

Table 3: Higher-Order Specs with Sugar

Spec	Ref'd Spec	Example
::symtab-id	::nat=	(symtab-id 42)
::value-attr	::bool	(value-attr false)
::dependencies	::identifier-set	(dependencies ['a 'b 'c])
::symbolic-value	TODO empty?	(symbolic-value ())
::value	TODO empty?	(value ())
::type-declaration	TODO nilable symtab-id	(type-declaration nil)
::varnym	::identifier	(varnym 'x)

Table 4: *Term* Multi-Specs

Multi-Spec	term	link	Sugared Conforming Example
::asr-term	::dimension	9.20	(dimension [6 60])
::asr-term	::intent	9.1	(intent 'Local)
::asr-term	::storage-type	9.4	(storage-type 'Default)
::asr-term	::abi	9.8	(abi 'Source)
::asr-term	::access	9.5	(access 'Public)
::asr-term	::presence	9.7	(presence 'Required)

Table 5: ::asr-term's with Nested *Head* Multi-Specs

term	head	link	Sugared Conforming Example
::ttype	::Integer	9.11	(Integer 4 [[6 60] [1 42]])
::ttype	::Real	9.11	(Real 8 [[6 60] [1 42]])
::ttype	::Complex	9.11	(Complex 4 [[6 60] [1 42]])
::ttype	::Logical	9.11	(Logical 1 [[6 60] [1 42]])
::symbol	::Variable	9.3.1	(Variable 42 'x (Integer 4)...) )

## 2.2 Multi-Specs

Hash-maps that conform to multi-spec `::asr-term` have a `::term` attribute that matches a defmethod. For instance, the following example is a valid `::asr-term`; its `::term` attribute is `::intent`:

```
(s/valid? ::asr-term
  {::term      ::intent, ;; matches a defmethod
   ::intent-enum 'Local}) ;; specifies contents
```

where

```
(s/def ::intent-enum
  #{'Local 'In 'Out 'InOut 'ReturnVar 'Unspecified})
```

The `::term`, `::intent`, matches the defmethod below. The keyword, `::term`, doubles as a key in the dictionary and as a tag for a defmulti and a defmethod.<sup>12</sup>

Here is another `::asr-term`, this time, matching `::abi`:

```
(s/valid? ::asr-term
  {::term      ::abi, ;; matches a defmethod
   ::abi-enum   'Source ;; specifies contents
   ::abi-external false}) ;; specifies contents
```

where

```
(s/def ::abi-external ::bool)
```

The other `::asr-term`'s follow the obvious pattern. The `::term` attributes, `::intent`, `::abi`, etc., match one of the term defmethod's:

```
(defmulti term ::term)
(defmethod term ::intent [_]
  (s/keys :req [::term ::intent-enum]))
(defmethod term ::dimension [_] ,,,)
(defmethod term ::abi [_] ,,,)
(defmethod term ::ttype [_] ,,,)
(defmethod term ::symbol [_] ,,,)
;; etc.
```

Finally, the multi-spec itself is named `::asr-term`:

```
;;      name of the multi-spec    defmulti tag fn
;;      -----
(s/def ::asr-term (s/multi-spec term ::term))
```

The multi-spec ties together the defmulti, the defmethod's, and the tag *qua* key `::term`. Specs that conform to `::asr-term` include *dimension*, *abi*, *ttype*, *symbol*, etc., in fact, all the terms in the ASDL grammar.<sup>8</sup>

<sup>12</sup><https://clojuredocs.org/clojure.core/defmulti>

## 2.3 Nested Multi-Specs

Contents of multi-specs can, themselves, be multi-specs. Such cases obtain when an `::asr-term` has multiple function-like heads. Examples include `::ttype`, `::symbol`, `::expr`, and `::stmt`.

The names of all multi-specs in MASR, nested or not, begin with `::asr-` and end with either `term` or `<some-term>-head`. Examples: `::asr-term` and `::asr-ttype-head`. There is only one level of nesting: terms above heads.

Here is the `::asr-term` spec for `::ttype`; the *nested* multi-spec for contents is named `::asr-ttype-head`.

```
(defmethod term ::ttype [_]
  (s/keys :req [::term ::asr-ttype-head]))
```

where

```
(defmulti ttype-head ::ttype-head)
(defmethod ttype-head ::Integer ,,,)
(defmethod ttype-head ::Real ,,,) ,,,
(s/def ::asr-ttype-head
  (s/multi-spec ttype-head ::ttype-head))
```

Here is a conforming example in full-form:

```
(s/valid? ::asr-term {::term ::ttype,
  ::asr-ttype-head
  {::ttype-head ::Real, ::real-kind 4,
   ::dimensions [[6 60] [1 42]]})
```

Likewise, here is the `::asr-term` spec for `::symbol`:

```
(defmulti symbol-head ::symbol-head)
(defmethod symbol-head ::Variable [_]
  (s/keys :req [::symbol-head ::syntab-id ::varnym ,,,]))
(defmethod symbol-head ::Module [_] ,,,)
(defmethod symbol-head ::Function [_] ,,,) ,,,
(s/def ::asr-symbol-head
  (s/multi-spec symbol-head ::symbolhead))
```

Here is a conforming example for `::Variable` in full-form, abbreviated:

```
(s/valid?
  ::asr-term {::term ::symbol,
  ::asr-symbol-head
  {::symbol-head ::Variable
   ::syntab-id (nat 2)
   ::varnym (identifier 'x)
   ::intent (intent 'Local)
   ::ttype (ttype (Integer 4 [[0 42]])) ,,, }})
```

## 2.4 Light Sugar, Heavy Sugar

*Light-sugar* forms are longer and more explicit than *heavy-sugar* forms, but shorter than full-form. Heavy-sugar forms require positional arguments, and are thus more brittle, especially for long specifications with many arguments, where the risk is high of getting them out of order in hand-written code.

Light-sugar specs have names with trailing hyphens. Light sugar specs typically have keyword arguments, partitioned into required and optional-with-defaults. The keyword argument lists of light-sugar specs do not depend on order. For example, the following conform to both `::asr-term` and to `::ttype`:

```
(ttype (Integer- {:dimensions [], :kind 4}))
(ttype (Integer- {:kind 4, :dimensions []}))
```

Heavy-sugar specs have names without trailing hyphens. Heavy-sugar specs are compatible with current ‘`libasr -show-asr`’ syntax. Heavy-sugar specs employ positional arguments that depend on order. Final arguments may have defaults. For example, the following conform to both `::asr-term` and to `::ttype`:

```
(ttype (Integer))
(ttype (Integer 4))
(ttype (Integer 2 []))
(ttype (Integer 8 [[6 60] [1 42]]))
```

Here is a conforming spec for `::Variable` in heavy sugar; it also conforms to `::asr-term`:

```
(let [a-valid (Variable 2 'x (Integer 4)
                        nil [] 'Local
                        [] [] 'Default
                        'Source 'Public 'Required
                        false)]
  (s/valid? ::asr-term a-valid) := true
  (s/valid? ::Variable a-valid) := true)
```

### 2.4.1 Term Entity-Key Specs

It is not convenient for terms to conform *only* to `::asr-term`. Therefore, we define redundant *term-entity-key* specs, like `::tterm`.

Entity-key specs for asr-terms are defined as follows:

```
(s/def ::ttype
  (s/and ::asr-term ;; must conform to ::asr-term
    ;; and have tag ::ttype
    # (= ::ttype (::term %))))
```

Because we have several such definitions, it is convenient to write a macro with a helper function:



```
(defn term-selector-spec [kwd]
  (s/and ::asr-term
    # (= kwd (::term %))))
(defmacro def-term-entity-key [term]
  (let [ns "masr.specs"
        tkw (keyword ns (str term))]
    `(s/def ~tkw ;; like ::tterm
      (term-selector-spec ~tkw))))
```

Here are some of the invocations of that macro:

```
(def-term-entity-key dimension)
(def-term-entity-key abi)
(def-term-entity-key ttype)
```

Here are some examples of extra conformance tests for sugared specs:

```
(s/valid? ::asr-term (dimension []))      := true
(s/valid? ::asr-term (dimension '(1 60))) := true
(s/valid? ::asr-term (dimension '()))     := true

(s/valid? ::dimension (dimension []))     := true
(s/valid? ::dimension (dimension '(1 60))) := true
(s/valid? ::dimension (dimension '()))     := true

(s/valid? ::asr-term (ttype (Integer 4))) := true
(s/valid? ::asr-term (ttype (Integer 4 []))) := true

(s/valid? ::ttype (ttype (Integer 4)))    := true
(s/valid? ::ttype (ttype (Integer 4 []))) := true
```

Entity-key specs are defined automatically for all *enum-like* terms via the `enum-like` macro:

```
(defmacro enum-like [term, heads]
  (let [ns "masr.specs"
        trm (keyword ns "term")      ;; like ::term
        art (keyword ns "asr-term")  ;; like ::asr-term
        tkw (keyword ns (str term))  ;; like ::intent
        '(),]
    `(do ;; the entity-key spec
      (s/def ~tkw ;; like ::intent
        (s/and ~art ;; like ::asr-term
          (term-selector-spec ~tkw)))
      '(),
    )))
```

Here are some examples of extra conformance tests for automatically defined enum-like specs:

```
(s/valid?  ::intent (intent 'Local)) := true

(let [iex (intent 'Local)]
  (s/conform ::asr-term iex)           := iex
  (s/conform ::intent iex)             := iex)
```

#### 2.4.2 Term-Head Entity-Key Specs

For terms like `::symbol` and `::stmt` with multiple heads like `::Variable` and `::Assignment`, respectively, it is convenient to define redundant entity-key specs like the following:

```
(s/def ::Variable           ;; head entity key
  (s/and ::asr-term         ;; top multi-spec
    # (= ::Variable        ;; nested tag
      (-> % ::asr-symbol-head ;; nested multi-spec
        ::symbol-head)))    ;; tag fetcher
  (s/def ::Assignment
    (s/and ::asr-term         ;; top multi-spec
      # (= ::Assignment      ;; nested tag
        (-> % ::asr-stmt-head ;; nested multi-spec
          ::stmt-head)       ;; tag fetcher
```

We define these with another macro, `def-term-head--entity-key`. The definition of this macro is found in the file `specs.clj`. An example of conformance to `::Variable` is found above, in Section 2.4.

We do not define term-head entity-key specs for every term, but only where convenient. For example, we don't define term-head entity-key specs for `::Integer`, `::Real`, `::Complex`, and `::Logical` `t`types. Conformance of such specs to `::ttype` suffice.

### 3 Issues with ASDL

Clojure solves the following issues with ASDL:

#### 3.1 ASDL is not Type-Checked

Type-checking for ASR instances written in ASDL is expressed only in hand-written C++ code. This situation is brittle. It's much better to have a specification language for ASR that expresses a type system.

#### 3.2 ASDL is Moribund

ASDL has not progressed since originally published in 1987. We know of no other projects adopting ASDL. We should replace ASDL with a modern metalanguage that has a robust, lively ecosystem.

#### 3.3 ASDL is Incomplete

Much of the semantics of ASR in ASDL, beyond type-checking, is expressed only in hand-written C++ code. The reason is that ASDL is not sufficiently expressive to cover the needed cases.

As usual with such a design, it's more time-consuming and error-prone than necessary to prototype, verify, validate, visualize, modify, and debug. Something more expressive than ASDL is needed to take some responsibility off of hand-written C++ code.

#### 3.4 ASDL's ASR is Volatile

The ASDL for ASR changes frequently, for good reasons. However, stand-aside tools like `asr-tester`<sup>13</sup> must chase the specification. Just keeping up with ASR-in-ASDL consumes almost all development time for `asr-tester`. We should unify the language that expresses ASR with the tools that verify and test ASR.

#### 3.5 ASDL is Ambiguous

There are many syntactic and semantic ambiguities in the ASDL grammar.<sup>8</sup> For example, the type notation `integer*` might mean, in one place, a list of `integer` with duplicate entries allowed, and, in another place, a set of `integer` with duplicate entries not allowed..

ASDL is not sufficient to express such distinctions. In practice, the hand-written C++ implementations implicitly make these distinctions, hiding them from view and making them difficult to revise. It is bad practice to hide fine distinctions that have observable effects in implementations. Instead, we should express those distinctions directly in the specifications.

Because ASDL cannot express such distinctions, we must adopt something more expressive than ASDL.

---

<sup>13</sup><https://github.com/rebcabin/asr-tester>

## 4 Clojure Solves ASDL Issues

### 4.1 Clojure Is a Complete Fit for ASR

ASR expressions, being trees, have a natural representation in S-Expressions.<sup>14</sup> Clojure, being a modern Lisp, natively handles S-Expressions.

### 4.2 Clojure is not Moribund

Clojure is up-to-date, lively, and production-ready.

### 4.3 Clojure Has Type-Checking Tools

Clojure.spec,<sup>15</sup> is a *force majeure* for precision, completeness, verification, and validation. The collection of MASR specs amounts to a meta-type system for ASR.

Clojure specs are arbitrary predicate functions. Clojure specs can easily express the difference between *list* and *set*, solving the ambiguity issue outlined in Section 3.5. Clojure specs, moreover, can flexibly express type-system features beyond the logics of typical, hard-coded type systems. That flexibility affords new long-term opportunities, say for experiments in dependent types and concurrency types.<sup>16</sup> In the short run, clojure.spec will make type constraints for ASDL explicit and manifest, and will relieve the burden on C++ programmers to manage implicit constraints.

### 4.4 ASR in Clojure Solves Volatility

We aim to replace ASDL with MASR in Clojure. When integrated with LCompilers in the future, MASR will be called LASR. There will be no gap or lag between ASR specs and their implementations.

---

<sup>14</sup><https://en.wikipedia.org/wiki/S-expression>

<sup>15</sup><https://clojuredocs.org/clojure.spec.alpha>

<sup>16</sup><https://rholang.io/>

## 5 MASR Definitions

**Definition 1.** A *spec* is a predicate function that tests an expression for conformance. *Spec* is a synonym for *type* in this document.

**Definition 2.** *Terms* are the "objects" or "productions" of ASR, like `symbol` or `dimension`.

Names of terms appear to the left of equals signs in the ASDL grammar.<sup>8</sup> Names of terms are generally in lower-case.

Table 6 exhibits terms, ambiguous types, and term-like types. Ambiguous types and term-like types are used but not defined in the ASDL grammar, but are explicitly defined in MASR.

The ambiguous types, `symbol_table` and `syntab_id`, are called out. The ASDL grammar conflates these two, having only `symbol_table` to mean either a full hash-map entity or an integer ID, depending on criteria hidden in hand-written C++ code. A primary objective of MASR is to remove this kind of ambiguity. This kind of ambiguity is not a deficiency of ASDL like that explained in Section 3.5. Unlike the difference between a list and a set, ASDL can express the difference between a hash-map and an integer ID. The failure to do so is a design flaw in the current ASDL grammar.

The contents of Table 6 have been greatly abbreviated and edited for presentation.

**Definition 3.** *Heads* are expressions like `Local` and `CaseStmt`, generally in `PascalCase`, that appear on the right-hand sides of equals signs in Table 6.

See the blog post in the footnote<sup>17</sup> for an informal description of *PascalCase*.

There are two kinds of heads:

*function-like heads* — have parentheses and typed parameters,  
e.g., `CaseStmt (expr*, stmt*)`

*enum-like heads* — no parentheses, e.g., `Local`

MASR has a Clojure spec and syntactic sugar for each head. There are about 250 heads by a recent count.

**Definition 4.** An *ASR entity* is a compound type like `CaseStmt (expr*, stmt*)`, with a function-like head and zero-or more arguments, possibly with names, that require recursive conformance.

<sup>17</sup><https://alok-verma6597.medium.com/case-styles-in-development-camel-pascal-snake-and-kebab-case-ed>

Table 6: Nodes in the ASDL Grammar

<b>term</b>	<b>partial expansion</b>
1 unit	TranslationUnit(symbol_table, node*)
2 symbol	...many heads...
3 storage_type	Default Save Parameter Allocatable
4 access	Public Private
5 intent	Local In Out InOut ...
6 deftype	Implementation Interface
7 presence	Required Optional
8 abi	Source LFortranModule ... Intrinsic
9 stmt	...many heads...
10 expr	...many heads...
11 ttype	Integer(int, dimension*) ...
12 restriction_arg	RestrictionArg( ident , symbol)
13 binop	Add Sub ... BitRShift
14 logicalbinop	And Or Xor NEqv Eqv
15 cmpop	Eq NotEq Lt LtE Gt GtE
16 integerboz	Binary Hex Octal
17 arraybound	LBound UBound
18 arraystorage	RowMajor ColMajor
19 cast_kind	RealToInteger IntegerToReal ...
20 dimension	(expr? start, expr? length)
21 alloc_arg	(expr a, dimension* dims)
22 attribute	Attribute( ident name, attr-arg* args)
23 attribute_arg	( ident arg)
24 call_arg	(expr? value)
25 tbind	Bind(string lang, string name)
26 array_index	( expr? left, expr? right, expr? step)
27 do_loop_head	( expr? v, expr? start expr? end, expr? step)
28 case_stmt	CaseStmt(expr*, stmt*) ...
29 type_stmt	TypeStmtName(symbol, stmt*) ...
30 enumtype	IntegerConsecutiveFromZero ...
<b>ambiguous</b>	
31 symbol_table	Clojure maps
32 symtab_id	int (new in MASR; not in ASDL)
<b>*term-like</b>	
0 dimensions	dimension*, via Clojure vectors or lists
0 atoms	int float bool nat bignat
0 identifier	by regex
0 identifiers	identifier*, via Clojure sets

## 6 MASR Tenets

**Entity Hash-Maps** — ASR entities<sup>11</sup> in full-form shall be hash-maps with fully-qualified keywords as keys (see Section 9.1 for motivating example).

**Multi-Specs** — ASR entity hash-maps shall be recursively checked and generated via Clojure multi-specs.<sup>11</sup>

**Full-Form** — Every entity has a fully explicit form in which all attributes are spelled out. Full-form hash-maps shall contain all necessary information, even at the cost of verbosity. Defaults and optionals appear only sparingly

**Syntax Sugar** — Certain constructor functions may supply default entity-attribute values so as to shorten common-case expressions. See Section 9.11 for an example and see Issue 3 on MASR's GitHub repo.<sup>18</sup>

---

<sup>18</sup><https://github.com/rebcabin/masr/issues/3>

## 7 Base Specs

The specs in this section are the *atoms* in Table 1 and atoms in the *term-like* grouping in Table 6.

### 7.1 Atoms: `int`, `float`, `bool`, `nat`

The specs for `int`, `float`, and `bool` are straightforward:

```
(s/def ::int int?) ;; java.lang.Long
(s/def ::float float?)
(s/def ::bool boolean?)
```

#### 7.1.1 Sugar

We restrict the spec, `nat`, for natural numbers, to `int`, for practical reasons:

```
(s/def ::nat nat-int?)
;; sugar
(defn nat [it]
  (let [cit (s/conform ::nat it)]
    (if (s/invalid? cit)
      ::invalid-nat
      cit)))
```

```
(tests
 (s/valid? ::nat (nat 42)) := true
 (s/valid? ::nat (nat -42)) := false
 (s/valid? ::nat (nat 0)) := true
 (s/valid? ::nat (nat 0xFFFFFFFFFFFFFFFF)) := false
 (s/valid? ::nat (nat -0xFFFFFFFFFFFFFFFF)) := false
 (s/valid?
  ::nat
  (nat (unchecked-long 0xFFFFFFFFFFFFFFFF))) := false
 (s/valid?
  ::nat
  (nat (unchecked-long -0xFFFFFFFFFFFFFFFF))) := true
 (s/valid? ::nat (nat 0x7FFFFFFFFFFFFFFFFF)) := true)
```



## 7.2 Notes

A Clojure *int* is a Java *Long*, with some peculiar behavior for hex literals.<sup>19</sup> The gist is that hex literals for negative numbers in Clojure must have explicit minus signs, lest they become `clojure.lang.BigInt`, which we disallow for dimension (9.20) and dimensions (8.1) in MASR. To get negative `java.lang.Long`, one employs Clojure's `unchecked-long`.

```
(tests (unchecked-long 0x8000000000000000)
      := -9223372036854775808
      (unchecked-long 0xFFFFFFFFFFFFFFFF)
      := -1
      (unchecked-long 0x8000000000000000)
      := -0x8000000000000000
      (unchecked-long -0xFFFFFFFFFFFFFFFF)
      := 1)
```

---

<sup>19</sup><https://clojurians.slack.com/archives/C03S1KBA2/p1681690965585429>

## 8 Term-Like Nodes

This section of the document exhibits specs for the *term-like nodes* in Tables 2 and 6: namely `dimensions` (plural), `identifier`, and `identifiers`. These are not terms, but share some similarities with terms. Note carefully the singulars and plurals in the names of the specs. `dimension` (singular) is a term and covered in Section 9.20.

### 8.1 `dimensions` [*sic*]

A MASR *dimensions* [*sic*], `dimension*` in ASDL, is a homogeneous ordered collection (list or vector) of zero or more `dimension` instances (9.20). Because `::dimensions` [*sic*] is not a term, we do not need nested multi-specs. However, because `::dimension` [*sic*] is a term, the elements of a `dimensions*` must conform to `::dimension`, which is an `asr-term` multi-spec. We ensure such conformance with a general-purpose function that selects terms that match a given spec, `dimension` in this case. We may reuse that function in other specs that represent non-term collections.

```
(defn term-selector-spec [kwd]
  (s/and ::asr-term
    #(= kwd (::term %))))
```

*Remark 1.* The notation `#(...%...)` is Clojure shorthand for an anonymous function (lambda) with a positional argument denoted by `%`, and positional arguments `%1`, `%2`, ... when there are two or more arguments. Applying a keyword like `::term` as a function picks that keyword out of its hash-map argument.

Here is the spec, `::dimensions`, for `dimensions`. We limit the number of `dimensions` to 9 for practical reasons. The meaning of an empty `::dimensions` instance is an open question (Issue 7<sup>20</sup>).

```
(def MIN-NUMBER-OF-DIMENSIONS 0) ;; TODO: 1?
(def MAX-NUMBER-OF-DIMENSIONS 9)

(s/def ::dimensions
  (s/coll-of (term-selector-spec ::dimension)
    :min-count MIN-NUMBER-OF-DIMENSIONS,
    :max-count MAX-NUMBER-OF-DIMENSIONS,
    :into []))
```

<sup>20</sup><https://github.com/rebcabin/masr/issues/7>

### 8.1.1 FullForm

The following tests show a couple of ways of writing out a `::dimensions` instance in full-form. The first is necessary in files other than `specs.clj`, say in `core_tests.clj`. The second can be used in `specs.clj`:

```
(tests (s/valid?
  ::dimensions
  [#:masr.specs{:term :masr.specs/dimension,
    :dimension-content [1 60]}
   #:masr.specs{:term :masr.specs/dimension,
    :dimension-content ()}]) := true
(s/valid?
  ::dimensions
  [{::term ::dimension,
    :dimension-content [1 60]}
   {::term ::dimension,
    :dimension-content ()}]) := true)
```

### 8.1.2 Sugar

The following tests illustrate the sugar for `::dimensions`:

```
(tests
  (s/valid? ::dimensions []) := true
  (s/valid? ::dimensions
    [(dimension '(1 60)) (dimension '())]) := true
  (s/conform ::dimensions
    [(dimension '(1 60)) (dimension '())]) :=
  [#:masr.specs{:term :masr.specs/dimension,
    :dimension-content [1 60]}
   #:masr.specs{:term :masr.specs/dimension,
    :dimension-content ()}])
```

## 8.2 identifier [sic]

An ASR identifier is a C or Fortran identifier, which begins with an alphabetic glyph or an underscore, and has alpha-numeric characters or underscores following. The only complication in the spec is the need to generate instances via `s/with-gen`. The spec solves the generation problem for identifiers, plus shows a pattern for other specs that need custom generators.

```
(let [alpha-re #"[a-zA-Z_]" ;; "let over lambda."
      alphameric-re #"[a-zA-Z0-9_]*"]
  (def alpha?
    #(re-matches alpha-re %))
  (def alphameric?
    #(re-matches alphameric-re %))
  (defn identifier? [sy]
    ;; exclude strings, numbers, quoted numbers
    (and (symbol? sy)
          (let [s (str sy)]
            (and (alpha? (subs s 0 1))
                  (alphameric? (subs s 1))))))
  (def identifier-generator
    (tgen/let [c (gen/char-alpha)
               s (gen/string-alphanumeric)]
              (symbol (str c s))))
  (s/def ::identifier
    (s/with-gen
      identifier?
      ;; fn wrapping a macro:
      (fn [] identifier-generator))))
```

The following tests illustrate validation and generation:

```
(tests
  (s/valid? :masr.specs/identifier 'foobar) := true
  (s/valid? :masr.specs/identifier '_f__547) := true
  (s/valid? :masr.specs/identifier '1234)    := false)
#_
(gen/sample (s/gen :masr.specs/identifier))
;; => (e c Q G Z2qP fXzg1 sRx2J6 YIhKlV k6 f7k1Xl4)
;; => (k hM LV QWC qW0X RGk3u W Kg6X Q2YvFO621 ODUt9)
```

### 8.2.1 Sugar

We define a simple function for creating conforming identifiers and illustrate it with a couple of tests:

```
(defn identifier [sym]
  (let [csym (s/conform ::identifier sym)]
    (if (s/invalid? csym)
        ::invalid-identifier
        csym)))
(tests
 (identifier 'foo)  := 'foo
 (identifier 123)   := ::invalid-identifier)
```

### 8.3 identifiers *[sic]*

ASDL `identifier*` is ambiguous. There are three kinds of identifier collections in MASR:<sup>21</sup>

**identifier-set** — unordered, no duplicates

**identifier-list** — ordered, duplicates allowed (we use vector)

**identifier-suit** — ordered, duplicates not allowed

For all three kinds, we limit the number of identifiers to 99 for practical purposes:

```
(def MIN-NUMBER-OF-IDENTIFIERS 0)
(def MAX-NUMBER-OF-IDENTIFIERS 99)
```

#### 8.3.1 identifier-set

The spec for a set of identifiers is straightforward because of Clojure's literal syntax, `#{\ldots}`, for sets, including the empty set:

```
(s/def ::identifier-set
  (s/coll-of ::identifier
    :min-count MIN-NUMBER-OF-IDENTIFIERS,
    :max-count MAX-NUMBER-OF-IDENTIFIERS,
    :into #{})) ;; empty set
```

See the code for uninteresting details of the sugar-function, `identifier-set`. The following tests show it at work:

```
(tests
  (let [x (identifier-set ['a 'a])]
    (s/valid? ::identifier-set x) := true
    (set? x) := true
    (count x) := 1)
  (let [x (identifier-set [])]
    (s/valid? ::identifier-set x) := true
    (set? x) := true
    (count x) := 0)
  (let [x (identifier-set ['a 'l])]
    (s/valid? ::identifier-set x) := false
    x := ::invalid-identifier-set))
```

---

<sup>21</sup><https://github.com/rebcabin/masr/issues/1>

### 8.3.2 identifier-list

The spec for a list of identifiers is almost the same as the spec for a set of identifiers. It differs only in the `:into` clause — into a vector rather than into a set:

```
(s/def ::identifier-list
  (s/coll-of ::identifier
    :min-count MIN-NUMBER-OF-IDENTIFIERS,
    :max-count MAX-NUMBER-OF-IDENTIFIERS,
    :into []))

(tests
  (every? vector? (gen/sample
    (s/gen ::identifier-list))) := true)
```

The implementation of the sugar-function for `identifier-list` is uninteresting. The following tests show it at work:

```
(tests
  (let [x (identifier-list ['a 'a])]
    (s/valid? ::identifier-list x) := true
    (vector? x) := true
    (count x) := 2)
  (let [x (identifier-list [])]
    (s/valid? ::identifier-list x) := true
    (vector? x) := true
    (count x) := 0)
  (let [x (identifier-list ['a '1])]
    (s/valid? ::identifier-list x) := false
    x := ::invalid-identifier-list))
```

### 8.3.3 identifier-suit

The spec for an identifier-suit is almost the same as for identifier-list, only checking that there are no duplicate elements

```
(s/def ::identifier-suit
  (s/and
    (s/coll-of ::identifier
      :min-count MIN-NUMBER-OF-IDENTIFIERS,
      :max-count MAX-NUMBER-OF-IDENTIFIERS,
      :into [])
    ;; no duplicates
    #(= (count %) (count (set %)))))
```

Here are the tests for the (uninteresting) sugar-function:

```
(tests
  (let [x (identifier-suit ['a 'a])]
    (s/valid? ::identifier-suit x) := false
    (vector? x) := false)
  (let [x (identifier-suit ['a 'b])]
    (s/valid? ::identifier-suit x) := true
    (vector? x) := true
    (count x) := 2)
  (let [x (identifier-suit [])]
    (s/valid? ::identifier-suit x) := true
    (vector? x) := true
    (count x) := 0)
  (let [x (identifier-suit ['a 'l])]
    (s/valid? ::identifier-suit x) := false
    x := ::invalid-identifier-suit))
```



## 9 Specs

The following sections

- summarize the Clojure specs for all ASR terms and heads (see Tables 4 and 5).
- pedagogically explain the architecture and approach taken in the Clojure code so that anyone may extend and maintain it.

The architecture is the remainder from several experiments. For example, `defrecord` and `defprotocol` for polymorphism were tried and discarded in favor of multi-specs.<sup>11</sup>

The tests in `core_test.clj` exhibit many examples that pass and, more importantly, fail the specs. We also keep lightweight, load-time tests inline to the source file for the specs, `specs.clj`. The balance between inline tests and separate tests is fluid.

The best way to learn the code is to study the tests and to run them in the Clojure REPL or in the CIDER debugger in Emacs.<sup>22</sup>

We present the terms somewhat out of the order of Table 6. First is *intent*, as it is the archetype for several enum-like terms and heads.

---

<sup>22</sup><https://docs.cider.mx/cider/debugging/debugger.html>

## 9.1 intent

### 9.1.1 Sets for Contents

An ASR *intent* is one of the symbols

Local, In, Out, InOut, ReturnVar, Unspecified.

The spec for the *contents* of an intent is simply this set of enum-like heads. Any Clojure *set* (e.g., in `#{ ... }` brackets) doubles as a predicate function for set membership. In the following two examples, the set appears in the function position of the usual Clojure function-call syntax (*function args\**):

If a candidate member is in a set, the result of calling the set like a function is the candidate member.

```
(#{'Local 'In 'Out 'InOut 'ReturnVar 'Unspecified} 'Local)
```

Local

When the candidate element, say *fubar*, is not in the set, the result is *nil*, which does not print:

```
(#{'Local 'In 'Out 'InOut 'ReturnVar 'Unspecified} 'fubar)
```

Any predicate function can be registered as a Clojure spec.<sup>15</sup> Therefore the spec for *intent contents* is just the set of valid members.

### 9.1.2 Specs have Fully Qualified Keyword Names

The name of the spec is `::intent-enum`. The double colon in `::intent-enum` is shorthand. In the file `specs.clj`, double colon implicitly signifies that a keyword like `intent-enum` is in the namespace `masr.specs`. In other files, like `core_test.clj`, the same keyword is spelled `:masr.specs/intent-enum`.

The names of all Clojure specs must be fully qualified in namespaces.

```
(s/def ::intent-enum
  #{'Local 'In 'Out 'InOut 'ReturnVar 'Unspecified})
```

### 9.1.3 How to Use Specs

To check an expression like `'Local` against the `::intent-enum` spec, write

```
(s/valid? ::intent-enum 'Local)
;; => true
(s/valid? ::intent-enum 'fubar)
;; => false
```

To produce conforming or non-conforming (invalid) entities in other code, write

```
(s/conform ::intent-enum 'Local)
;; => Local
(s/conform ::intent-enum 'fubar)
;; => :clojure.spec.alpha/invalid
```

To generate a few conforming samples, write

```
(gen/sample (s/gen ::intent-enum) 5)
;; => (Unspecified Unspecified Out Unspecified Local)
```

or, with conformance explanation (trivial in this case):

```
(s/exercise ::intent-enum 5)
;; => ([Out Out]
;;      [ReturnVar ReturnVar]
;;      [In In]
;;      [Local Local]
;;      [ReturnVar ReturnVar])
```

Strip out the conformance information as follows:

```
(map second (s/exercise ::intent-enum 5))
;; => (In ReturnVar Out In ReturnVar)
```

`s/valid?`, `s/conform`, `gen/sample`, and `s/exercise` pertain to any Clojure specs, no matter how complex or rich.

### 9.1.4 The Spec that Contains the Contents

`::intent-enum` is just the spec for the *contents* of an intent, not for the intent itself. The spec for the intent itself is an implementation of a polymorphic Clojure *multi-spec*,<sup>11</sup> `::asr-term`.

### 9.1.5 Multi-Specs

A multi-spec is like a tagged union in C. The multi-spec, `::asr-term`, pertains to all Clojure hash-maps<sup>23</sup> that have a tag named `::term` with a value like `::intent` or `::storage-type`, etc. The values, if themselves fully qualified keywords, are recursively checked.

A multi-spec has three components:

**defmulti**<sup>24</sup> — a polymorphic interface that declares the *tag-fetcher function*, `::term` in this case. The tag-fetcher function fetches a tag's value from any candidate hash-map. The `defmulti` dispatches to a `defmethod` that matches the fetched tag value, `::intent` in this case. `::term` is a fully qualified keyword of course, but all keywords double as tag-fetchers for hash-maps.<sup>25</sup>

**defmethod**<sup>26</sup> — individual specs, each implementing the interface; in this case, if the `::term` of a hash-map matches `::intent`, then the corresponding `defmethod` is invoked (see Section 9.1.7 below).

**s/multi-spec** — tying together the `defmulti` and, redundantly, the tag-fetcher.<sup>27</sup>

### 9.1.6 Specs for All Terms

Start with a spec for `::term`:

```
;; like ::intent, ::symbol, ::expr, ...  
(s/def ::term qualified-keyword?)
```

The spec says that any fully qualified keyword, like `::intent`, is a MASR term. This spec leaves room for growth of MASR by adding more fully qualified keywords for more MASR types-*qua*-terms.

`s/def` stands for `clojure.spec.alpha/def`, the `def` macro in the `clojure.spec.alpha` namespace. The namespace is aliased to `s`.

Next, specify the `defmulti` polymorphic interface, `term`, (no colons) for all term specs:

```
(defmulti term ::term)
```

This `defmulti` dispatches to a `defmethod` based on the results of applying the keyword-*qua*-function `::term` to a hash-map:

```
(::term {::term ::intent ...})
```

`equals ::intent`.

<sup>23</sup><https://clojuredocs.org/clojure.core/hash-map>

<sup>24</sup><https://clojuredocs.org/clojure.core/defmulti>

<sup>25</sup><https://stackoverflow.com/questions/6915531>

<sup>26</sup><https://clojuredocs.org/clojure.core/defmethod>

<sup>27</sup>Multi-specs allow re-tagging, but we do not need that level of generality.

The spec is named `::term` and the tag-fetcher is named `::term`. They don't need to be the same. They could have different names.

### 9.1.7 Spec for intent

If applying `::term` to a Clojure hash-map produces `::intent`, the following spec, which specifies all intents, will be invoked. It ignores its argument, `_`:

```
(defmethod term ::intent [_]
  (s/keys :req [::term ::intent-enum]))
```

This spec states that an *intent* is a Clojure hash-map with a `::term` keyword and an `::intent-enum` keyword.

### 9.1.8 The Multi-Spec Itself: `::asr-term`

`s/multi-spec` ties `defmulti term` to the tag-fetcher `::term`. The multi-spec itself is named `::asr-term`:

```
;;      name of the multi-spec    defmulti tag fn
;;      -----
(s/def ::asr-term (s/multi-spec term ::term))
```

### 9.1.9 Examples of Intent

The following shows a valid example:

```
(s/valid? ::asr-term
  {::term      ::intent,
   ::intent-enum 'Local})
```

true

Here is an invalid sample:

```
(s/valid? ::asr-term
  {::term      ::intent,
   ::intent-enum 'FooBar})
```

false

Generate a few valid samples:

```
(gen/sample (s/gen (s/and
                    ::asr/asr-term
                    #(= ::asr/intent (::asr/term %))))
            5)
;;=> (::asr{:term ::asr/intent, :intent-enum ReturnVar}
;;    (::asr{:term ::asr/intent, :intent-enum In}
;;    (::asr{:term ::asr/intent, :intent-enum Unspecified}
;;    (::asr{:term ::asr/intent, :intent-enum Unspecified}
;;    (::asr{:term ::asr/intent, :intent-enum InOut})))
```

#### 9.1.10 Another asr-term: a Pattern Emerges

To define another asr-term, specify the contents and write a defmethod. The one multi-spec, `::asr-term`, suffices for all.

For example, another asr-term for an enum-like is `storage-type`:

```
(s/def ::storage-type-enum
      #{'Default, 'Save, 'Parameter, 'Allocatable})

(defmethod term ::storage-type [_]
  (s/keys :req [::term ::storage-type-enum]))
```

All enum-like specs follow this pattern.

#### 9.1.11 Syntax Sugar

`{::term ::intent, ::intent-enum 'Local}`, a valid asr-term entity, is long and ugly. Write a short function, `intent`, via `s/conform`, explained in Section 9.1.3:

```
(defn intent [sym]
  (let [intent_ (s/conform
                  ::asr-term
                  {::term ::intent, ::intent-enum sym})]
    (if (s/invalid? intent_)
        ::invalid-intent
        intent_)))
```

Entities have shorter expression with the sugar:

```
(testing "better syntax"
  (is (s/valid? ::asr-term (intent 'Local)))
  (is (s/valid? ::asr-term (intent 'Unspecified)))
  (is (not (s/valid? ::asr-term (intent 'foobar))))
  (is (not (s/valid? ::asr-term (intent []))))
  (is (not (s/valid? ::asr-term (intent ())))
  (is (not (s/valid? ::asr-term (intent {}))))
  (is (not (s/valid? ::asr-term (intent #{ }))))
  (is (not (s/valid? ::asr-term (intent "foobar"))))
  (is (not (s/valid? ::asr-term (intent ""))))
  (is (not (s/valid? ::asr-term (intent 42))))
  (is (thrown? clojure.lang.ArityException (intent))))
```

All our specs are like that: a long-form hash-map and a short-form sugar function that does a conformance check.

### 9.1.12 Capture the Enum-Like Pattern in a Macro

All enum-likes have a *contents* spec, a `defmethod term`, and a syntax-sugar function. The following macro pertains to all such enum-like multi-specs:

```
(defmacro enum-like [term, heads]
  (let [ns "masr.specs"
        tkw (keyword ns (str term))
        tke (keyword ns (str term "-enum"))
        tki (keyword ns (str "invalid-" term))]
    `(do
      (s/def ~tke ~heads) ;; the set
      (defmethod term ~tkw [_#] ;; the multi-spec
        (s/keys :req [:masr.specs/term ~tke]))
      (defn ~term [it#] ;; the syntax
        (let [st# (s/conform
                    :masr.specs/asr-term
                    {:masr.specs/term ~tkw
                     ~tke it#})]
          (if (s/invalid? st#) ~tki, st#))))))
```

Use the macro like this:

```
(enum-like
 intent
 #{'Local 'In 'Out 'InOut 'ReturnVar 'Unspecified})
(enum-like
 storage-type
 #{'Default, 'Save, 'Parameter, 'Allocatable})
```

## 9.2 **TODO** unit



## 9.3 TODO symbol

### 9.3.1 TODO Variable

Here is an example of the full form for a `::Variable` with conforming instances in light sugar (Section 2.4):

```
(let [a-var-head
      {::symbol-head ::Variable
       ::syntab-id   (nat 2)
       ::varnym      (varnym 'x)
       ::ttype       (ttype (Integer 4 []))
       ::type-declaration (type-declaration nil)
       ::dependencies (identifier-set ())
       ::intent       (intent 'Local)
       ::symbolic-value () ;; TODO sugar
       ::value        () ;; TODO sugar
       ::storage-type (storage-type 'Default)
       ::abi           (abi 'Source :external false)
       ::access        (access 'Public)
       ::presence      (presence 'Required)
       ::value-attr    false ;; TODO sugar
      }
      a-var {::term ::symbol
             ::asr-symbol-head a-var-head}
      a-var-light (Variable-
                   :varnym      (identifier 'x)
                   :syntab-id   2
                   :ttype       (ttype (Integer 4)))])
```

```
(tests
  a-var-light := (s/conform ::asr-term a-var)
  a-var-light := (s/conform ::Variable a-var)

  (s/valid? ::asr-symbol-head a-var-head) := true

  (s/valid? ::asr-term a-var)             := true
  (s/valid? ::asr-term a-var-light)       := true

  (s/valid? ::Variable a-var)             := true
  (s/valid? ::Variable a-var-light)       := true
))
```

## 9.4 storage\_type

The following tests illustrate conformance:

```
(tests
  (s/valid? ::storage-type-enum 'Default)      := true
  (s/valid? ::storage-type-enum 'foobar)       := false
  (s/valid? ::asr-term
    {::term ::storage-type
      ::storage-type-enum 'Default})          := true
  (s/valid? ::asr-term (storage-type 'Default)) := true
  (s/valid? ::asr-term (storage-type 'foobar)) := false
  (storage-type 'foobar)
    := ::invalid-storage-type
  (s/conform ::asr-term
    (storage-type 'Default)
    := (storage-type 'Default)))
```

The implementation is covered in Section 9.1.12 on the enum-like macro.

## 9.5 access

```
(enum-like access #{'Public 'Private})  
(tests  
  (let [public (access 'Public)]  
    (s/conform ::asr-term public) := public)  
  (access 'foobar) := ::invalid-access)
```

## 9.6 TODO deftype

## 9.7 presence

```
(enum-like presence #{'Required' 'Optional'})  
(tests  
  (let [required (presence 'Required')]  
    (s/conform ::asr-term required) := required)  
  (presence 'fubar) := ::invalid-presence)
```

## 9.8 abi

*Abi* is a rich case. It is enum-like, similar to *intent* (Section 9.1), but with restrictions. Its heads include several *external-abis*:

```
(def external-abis
  #{ 'LFortranModule, 'GFortranModule,
    'BindC, 'Interactive, 'Intrinsic})
```

and one *internal-abi*, specified as a Clojure set to get the membership-test functionality:

```
(def internal-abis #{'Source})
```

The *abi-enum* spec for the contents of an *abi* term is the unions of these two sets:

```
(s/def ::abi-enum
  (set/union external-abis internal-abis))
```

Specify an additional key in a conforming *abi* hash-map with a `::bool` predicate:

```
(s/def ::abi-external ::bool)
```

Add a convenience function for logic:

```
(defn iff [a b]
  (or (and a b)
      (not (or a b))))
```

Specify the `defmethod` for the *abi* itself with a hand-written generator (clojure.spec is not quite strong enough to create the generator automatically):

```
(defmethod term ::abi [_]
  (s/with-gen
    (s/and
      #(iff (= 'Source (::abi-enum %))
            (not (::abi-external %)))
      (s/keys :req [::term ::abi-enum ::abi-external]))
    (fn []
      (tgen/one-of
        [(tgen/hash-map
          ::term (gen/return ::abi)
          ::abi-enum (s/gen external-abis)
          ::abi-external (gen/return true))
         (tgen/hash-map
          ::term (gen/return ::abi)
          ::abi-enum (s/gen internal-abis)
          ::abi-external (gen/return false))] ))))
```

Generate a few conforming samples:

```
(gen/sample (s/gen (s/and
                  ::asr/asr-term
                  #(= ::asr/abi (::asr/term %))))
            5)
;; => (::asr{:term ::asr/abi,
;;       :abi-enum Interactive, :abi-external true}
;;     #::asr{:term ::asr/abi,
;;       :abi-enum Source, :abi-external false}
;;     #::asr{:term ::asr/abi,
;;       :abi-enum Source, :abi-external false}
;;     #::asr{:term ::asr/abi,
;;       :abi-enum Source, :abi-external false}
;;     #::asr{:term ::asr/abi,
;;       :abi-enum Source, :abi-external false}
;;     #::asr{:term ::asr/abi,
;;       :abi-enum Interactive, :abi-external true})
```

### 9.8.1 Syntax Sugar

The light-sugar for *abi* uses Clojure destructuring<sup>28,29</sup> for keyword arguments.

Conforming examples:

```
(abi 'Source      :external false)
(abi 'LFortranModule :external true)
(abi 'GFortranModule :external true)
(abi 'BindC       :external true)
(abi 'Interactive  :external true)
(abi 'Intrinsic   :external true)
```

Non-conforming due to incorrect boolean:

```
(abi 'Source      :external true)
(abi 'LFortranModule :external false)
(abi 'GFortranModule :external false)
(abi 'BindC       :external false)
(abi 'Interactive  :external false)
(abi 'Intrinsic   :external false)
```

<sup>28</sup><https://clojure.org/guides/destructuring>

<sup>29</sup><https://gist.github.com/rebcabin/a3c24be3e17135f355348c834ab14141>

Non-conforming due to incorrect types or structure:

```
(abi 'Source :external 42)      ;; types are not ::bool
(abt 'Source :external "foo")  ;; /
(abt 'Source :external 'foo)   ;; ==-
(abt 'Source false)           ;; no :external keyword
(abt 'Source true)            ;; /
(abt 'Source 42)               ;; /
(abt 'foo true)                ;; /
(abt 'foo false)               ;; ==-
```

We don't show tests of incorrect arity.

Here is the implementation of the sugar, exhibiting the destructuring technique:

```
(defn abi
  "Destructure the keyword :external"
  [the-abi-enum, & {:keys [external]}]
  (let [abi_ (s/conform
    ::asr-term
    {::term ::abi,
     ::abi-enum the-abi-enum,
     ::abi-external external})]
    (if (s/invalid? abi_)
      ::invalid-abi
      abi_)))
```

## 9.9 **TODO** stmt



## 9.10 **TODO** expr

## 9.11 ttype

Ttype [sic] features a nested multi-spec. Ttype is an archetype for all function-like heads, just as *intent* is an archetype for all enum-like heads.

```
(defmulti ttype-head ::ttype-head)
(defmethod ttype-head ::Integer [_]
  (s/keys :req [::ttype-head ::bytes-kind ::dimensions]))
(s/def ::asr-ttype-head
  (s/multi-spec ttype-head ::ttype-head))
```

```
(defmethod term ::ttype [_]
  (s/keys :req [::term ::asr-ttype-head]))
```

### 9.11.1 Full Form

One may always write out ttype specs in full:

```
(s/valid? ::asr-term
  {::term ::ttype,
   ::asr-ttype-head
   {::ttype-head ::Integer,
    ::bytes-kind 4
    ::dimensions [[6 60] [1 82]]}})
```

### 9.11.2 Sugar for Integer, Real, Complex, Logical

Sugar for ttypes comes in two varieties, *light sugar* and *heavy sugar*. See Section 2.4 for rationale.

#### 1. Light Sugar Examples

```
(ttype (Integer- {:dimensions [], :kind 4}))
(ttype (Integer- {:kind 4, :dimensions []}))
```

#### 2. Heavy Sugar Examples

```
(ttype (Integer))
(ttype (Integer 4))
(ttype (Integer 2 []))
(ttype (Integer 8 [[6 60] [1 42]]))
```

See the tests for many more examples.

### 9.11.3 TODO Character

## 9.12 TODO restriction\_arg

### 9.13 **TODO** binop

## 9.14 **TODO** logicalbinop

## 9.15 **TODO** cmpop

## 9.16 **TODO** integerboz

## 9.17 **TODO** arraybound

## 9.18 **TODO** arraystorage



## 9.19 **TODO** cast\_kind

## 9.20 dimension

A *dimension* is 0 or 2 nats in a Clojure list or vector:

```
(def MIN-DIMENSION-COUNT 0)
(def MAX-DIMENSION-COUNT 2)
(s/def ::dimension-content
  (s/and (fn [it] (not (= 1 (count it))))
    (s/coll-of ::nat
      :min-count MIN-DIMENSION-COUNT,
      :max-count MAX-DIMENSION-COUNT,
      :into ())))
```

If there are two nats, the first nat specifies the starting index of any array dimension that enjoys the instance, and the second nat specifies the length. For example, in the ttype `(Integer 4 [[6 60]])` (9.11), the one dimension in the dimensions `[sic]` (8.1) of the ttype is `[6 60]`. The ttype specifies a rank-1 array of 60 4-byte integers with indices starting at 6 and running through 65.

If there are no nats, i.e., the array dimension of any array enjoying the instance is of zero length. For an example, consider the ttype `(Integer 4 [[]])` (9.11). This meaning of this type is an open question.<sup>20</sup>

### 9.20.1 Empty Dimensions

Empty dimensions `[sic]`, as in `(Integer 4 [])` specify non-array types. These are often called, loosely, *scalars*. Pedantically, the term *scalar* only pertains with respect to a vector space.

An empty dimension, as in `(Integer 4 [[]])`, specifies a rank-1 array of zero length. Such items are discussed further in Issue 7 and in Section 8.1.

### 9.20.2 TODO: Issue 7: Zero Length

The following specs, in context of a ttype (9.11) for convenience, are legal in the ASDL grammar.<sup>8</sup> They all denote arrays of length 0, and the meaning of an array of length 0 is **unspecified**:

```
(Integer 4 [[]])
(Integer 4 [[0]])
(Integer 4 [[6 0]])
```

See Issue 7 in MASR's GitHub repo.<sup>20</sup>

### 9.20.3 FullForm

The following tests illustrate the full form for *dimension*:

```
(tests
  (s/valid? ::asr-term
    {::term ::dimension
      ::dimension-content [6 60]}) := true
  (s/valid? ::asr-term
    {::term ::dimension
      ::dimension-content [0]})   := false
  (s/valid? ::asr-term
    {::term ::dimension
      ::dimension-content []})    := true)
```

### 9.20.4 Sugar

The following tests illustrate the syntactic sugar for *dimension*:

```
(tests
  (s/conform ::asr-term
    {::term ::dimension,
      ::dimension-content '(1 60)}) :=
  (dimension '(1 60))
  (s/valid? ::asr-term (dimension 60))           := false
  (s/valid? ::asr-term (dimension [[]]))          := false
  (s/valid? ::asr-term (dimension 'foobar))        := false
  (s/valid? ::asr-term (dimension ['foobar']))     := false
  ;; throw arity (s/valid? ::asr-term (dimension)) := false
  (s/valid? ::asr-term (dimension []))             := true
  (s/valid? ::asr-term (dimension [60]))           := false
  (s/valid? ::asr-term (dimension [0]))            := false
  (s/valid? ::asr-term (dimension '(1 60)))        := true
  (s/valid? ::asr-term (dimension ' ()))           := true)
```

## 9.21 **TODO** alloc\_arg

## 9.22 **TODO** attribute

## 9.23 **TODO** attribute\_arg

## 9.24 **TODO** call\_arg

## 9.25 **TODO** tbind



## 9.26 **TODO** array\_index

## 9.27 **TODO** do\_loop\_head

## 9.28 **TODO** case\_stmt

## 9.29 **TODO** type\_stmt

### 9.30 **TODO** enumtype

## 10 Implicit Terms

Terms used, explicitly or implicitly, but not defined in ASDL.

Some items specified in ASDL as *symbol\_table* are actually *syntab\_id*.

### 10.1 syntab\_id

```
(s/def ::syntab-id ::nat)
```

## 10.2 TODO symbol\_table

```
(s/def ::symbol-table map?
```