

MASR — Meta Abstract Semantics Representation

Brian Beckman

23 Apr 2023

Contents

1	Abstract	3
2	Issues with ASDL	4
2.1	ASDL is not Type-Checked	4
2.2	ASDL is Moribund	4
2.3	ASDL is Incomplete	4
2.4	ASDL's ASR is Volatile	4
2.5	ASDL is Ambiguous	4
3	Clojure Solves ASDL Issues	5
3.1	Clojure Is a Complete Fit for ASR	5
3.2	Clojure is not Moribund	5
3.3	Clojure Has Type-Checking Tools	5
3.4	ASR in Clojure Solves Volatility	5
4	MASR Definitions	6
5	MASR Tenets	9
6	Base Specs	10
6.1	Atoms: <code>int</code> , <code>float</code> , <code>bool</code> , <code>nat</code>	10
6.2	Notes	11
7	Term-Like Nodes	12
7.1	dimensions [<i>plural</i>]	12
7.2	identifier [<i>singular</i>]	14
7.3	identifiers [<i>plural</i>]	16
8	Specs	19
8.1	intent	20
8.2	TODO unit	27
8.3	TODO symbol	28
8.4	storage_type	30

8.5	access	31
8.6	TODO deftype	31
8.7	presence	31
8.8	abi	32
8.9	TODO stmt	36
8.10	TODO expr	37
8.11	ttype	38
8.12	TODO restriction_arg	38
8.13	TODO binop	39
8.14	TODO logicalbinop	40
8.15	TODO cmpop	41
8.16	TODO integerboz	42
8.17	TODO arraybound	43
8.18	TODO arraystorage	44
8.19	TODO cast_kind	45
8.20	dimension	46
8.21	TODO alloc_arg	48
8.22	TODO attribute	49
8.23	TODO attribute_arg	50
8.24	TODO call_arg	51
8.25	TODO tbind	52
8.26	TODO array_index	53
8.27	TODO do_loop_head	54
8.28	TODO case_stmt	55
8.29	TODO type_stmt	56
8.30	TODO enumtype	57
9	Implicit Terms	58
9.1	symtab_id	58
9.2	TODO symbol_table	59

1 Abstract

Abstract Semantics Representation (ASR) is a novel intermediate representation (IR)¹ for a new collection of LCompilers [sic].² ASR is agnostic to the particular programming language under compilation. Current compiler front-ends targeting ASR include LFortran³ and LPython.⁴ ASR is also agnostic to the back end. ASR currently targets LLVM, x86, C, and WASM⁵

Typical IRs encode semantics as decorations on the Abstract Syntax Tree, (AST)⁶ ASR lifts *semantics* to the top level and expunges the syntax of the surface language as early as possible. Free of syntactical baggage, ASR optimizers are cleaner and faster than average, and ASR back ends are completely reusable. If syntax information is ever necessary, as with semantical-feedback parsing, such information will be encoded as decorations on the ASR, rather than the other way around.

Current specifications for ASR are written in ASDL,⁷ a metalanguage similar in spirit to yacc but less rich, by design.⁸ To build an LCompiler like LFortran or LPython, the ASDL grammar⁹ for ASR is parsed, and a library in C++, libasr,¹⁰ is generated. Compiler front ends call functions in this library to manipulate ASR and to emit code from the back ends.

ASDL has several deficiencies, and MASR¹¹ alleviates them. Chief among the deficiencies is the lack of type-checking. MASR adds a type system to ASR via Clojure *specs*.¹² MASR is a complete programming language in its own regard. It is, in fact, a Domain-Specific Language (DSL),¹³ embedded in Clojure.¹⁴

We aim to replace ASDL with MASR and to integrate MASR with the LCompiler code base. When so 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 libasr, at least until MASR replaces ASDL. The document mirrors an ASDL snapshot.⁹

¹https://en.wikipedia.org/wiki/Intermediate_representation

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

³<https://lfortran.org/>

⁴<https://lpython.org/>

⁵<https://webassembly.org/>

⁶https://en.wikipedia.org/wiki/Abstract_syntax_tree

⁷https://en.wikipedia.org/wiki/Abstract-Type_and_Scheme-Definition_Language

⁸<https://en.wikipedia.org/wiki/Yacc>

⁹https://github.com/rebcabin/masr/blob/main/ASR_2023_APR_06_snapshot.asdl

¹⁰<https://github.com/lfortran/lfortran/tree/c648a8d824242b676512a038bf2257f3b28dad3b/src/libasr>

¹¹pronounced “maser;” it is a Physics pun

¹²<https://clojure.org/guides/spec>

¹³https://en.wikipedia.org/wiki/Domain-specific_language

¹⁴<https://en.wikipedia.org/wiki/Clojure>

2 Issues with ASDL

Clojure solves the following issues with ASDL:

2.1 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.

2.2 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.

2.3 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.

2.4 ASDL's ASR is Volatile

The ASDL for ASR changes frequently, for good reasons. However, stand-aside tools like `asr-tester`¹⁵ 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.

2.5 ASDL is Ambiguous

There are many syntactic and semantic ambiguities in the ASDL grammar.⁹ 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.

¹⁵<https://github.com/rebcabin/asr-tester>

3 Clojure Solves ASDL Issues

3.1 Clojure Is a Complete Fit for ASR

ASR expressions, being trees, have a natural representation in S-Expressions.¹⁶ Clojure, being a modern Lisp, natively handles S-Expressions.

3.2 Clojure is not Moribund

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

3.3 Clojure Has Type-Checking Tools

Clojure.spec,¹⁷ 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 2.5. 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.¹⁸ 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.

3.4 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 LASR specs and their implementations because the implementations will be generated at build time.

¹⁶<https://en.wikipedia.org/wiki/S-expression>

¹⁷<https://clojuredocs.org/clojure.spec.alpha>

¹⁸<https://rholang.io/>

4 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 snapshot.⁹ Names of terms are generally in lower-case.

Table 1 exhibits terms, ambiguous types, and term-like types, which are used-but-not-defined in the ASDL grammar. MASR explicitly defines them. Each term has a Clojure spec, various sugar functions, and term-entity key specs (see the companion document, `masr.summary.pdf`).

The ambiguous types, `symbol_table` and `syntab_id`, are called out. The ASDL grammar conflates these two, having only `symbol_table`, which can either a full hash-map entity or an integer ID, depending on obscure criteria hidden in hand-written C++ code. A primary objective of MASR is to remove this kind of ambiguity, which is a design flaw in the current ASDL grammar rather than a deficiency of ASDL because ASDL can express the difference between a hash-map and an integer ID.

The contents of Table 1 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 1.

See the blog post in the footnote¹⁹ 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 sugar functions for each head. Most heads also have term-head entity-key specs (see the companion document, `masr-summary.pdf`). There are about 250 heads by a recent count.

Definition 4. An *ASR entity* is a compound type like

¹⁹<https://alok-verma6597.medium.com/case-styles-in-development-camel-pascal-snake-and-kebab-case-ed>

`CaseStmt (expr*, stmt*)`, with a function-like head and zero-or more arguments, possibly with names, that require recursive conformance.

Table 1: Nodes in the ASDL Grammar

term	partial expansion
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 ...
ambiguous	
31 symbol_table	Clojure maps
32 symtab_id	int (new in MASR; not in ASDL)
*term-like	
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

5 MASR Tenets

See the companion document, `masr-summary.pdf`, for much discussion of the terminology here.

Elegance — MASR is elegant, beautiful, thus easy to learn, understand, and maintain.

Entity Hash-Maps — ASR entities¹² in full-form shall be hash-maps with fully-qualified keywords as keys for a summary and Section 8.1 for a motivating example, *intent*).

Full-Form — Every entity shall have 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.

Multi-Specs — ASR entity hash-maps shall be recursively checked and generated via Clojure multi-specs. See the body of this document for examples.

Syntax Sugar — Certain constructor functions may supply default entity-attribute values so as to shorten common-case expressions. See Section 8.11 for an example, and see Issue 3 on MASR's GitHub repo.²⁰

²⁰<https://github.com/rebcabin/masr/issues/3>

6 Base Specs

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

6.1 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?)
```

6.1.1 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 0xFFFFFFFF))   := false
 (s/valid? ::nat (nat -0xFFFFFFFF))  := false
 (s/valid?
  ::nat
  (nat (unchecked-long 0xFFFFFFFF))) := false
 (s/valid?
  ::nat
  (nat (unchecked-long -0xFFFFFFFF))) := true
 (s/valid? ::nat (nat 0x7FFFFFFFF))  := true)
```

6.2 Notes

A Clojure *int* is a Java *Long*, with some peculiar behavior for hex literals.²¹ Hex literals for negative numbers in Clojure must have explicit minus signs, lest they become `clojure.lang.BigInt`. MASR disallows `BigInt` for dimension (8.20) and dimensions (7.1). To get negative `java.lang.Long` without explicit minus signs, one employs Clojure's `unchecked-long`.

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

²¹<https://clojurians.slack.com/archives/C03S1KBA2/p1681690965585429>

7 Term-Like Nodes

This section of the document exhibits specs for the *term-like nodes* in Table 1: 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 8.20, whereas `dimensions` (plural) is not a term. In the ASDL snapshot grammar,⁹ the plural of `dimension` is denoted `dimension*`, with a Kleene star.

7.1 dimensions [plural]

A MASR *dimensions* [plural], `dimension*` in ASDL, is a homogeneous ordered collection (list or vector) of zero or more `dimension` instances (8.20). Because `::dimensions` [plural] is not a term, we do not need nested multi-specs. However, because `::dimension` [singular] 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. MASR reuses 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 7²²).

```
(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 []))
```

²²<https://github.com/rebcabin/masr/issues/7>

7.1.1 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)
```

7.1.2 Sugar

The following tests illustrate the sugar function, `dimensions`, for the spec, `::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 ()}])
```

7.2 identifier [*singular*]

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. 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)
```

7.2.1 Sugar

We define and illustrate the sugar function, `identifier` for creating identifiers:

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

7.3 identifiers [*plural*]

ASDL `identifier*` is ambiguous. There are three kinds of identifier collections in MASR:²³

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)
```

7.3.1 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))
```

²³<https://github.com/rebcabin/masr/issues/1>

7.3.2 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))
```

7.3.3 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))
```

8 Specs

The following sections

- summarize the Clojure specs for all ASR terms and heads.
- 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.

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, `specs.clj`. We don't have strict criteria for whether a test should be inline, separate, or both.

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.²⁴

We present the terms somewhat out of the order of Table 1. First is *intent*, as it is the archetype for several enum-like terms and heads.

²⁴<https://docs.cider.mx/cider/debugging/debugger.html>

8.1 intent

8.1.1 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.¹⁷ Therefore the spec for *intent contents* is just the set of valid members.

8.1.2 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})
```

8.1.3 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.

8.1.4 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*,¹² `::asr-term`.

8.1.5 Multi-Specs

A multi-spec is like a tagged union in C. The multi-spec, `::asr-term`, pertains to all Clojure hash-maps²⁵ 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`[[fn:defmulti:

<https://clojuredocs.org/clojure.core/defmulti> :: — 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.²⁶

defmethod²⁷ — 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 8.1.7 below).

s/multi-spec — tying together the `defmulti` and, redundantly, the tag-fetcher.²⁸

8.1.6 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, and bind it to the tag-fetcher, `::term`:

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

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

```
(::term {::term ::intent ...})  
;; => ::intent
```

²⁵<https://clojuredocs.org/clojure.core/hash-map>

²⁶<https://stackoverflow.com/questions/6915531>

²⁷<https://clojuredocs.org/clojure.core/defmethod>

²⁸Multi-specs allow re-tagging, where the tag named in the multi-spec is different from the tag-fetcher function. MASR does not need re-tagging.

The spec is named `::term` and the tag-fetcher is named `::term`. They don't need to be the same name, but they always are in MASR.

8.1.7 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 required `::term` keyword and a required `::intent-enum` keyword. This kind of spec is called an *entity spec*.¹²

8.1.8 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))
```

8.1.9 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})))
```

8.1.10 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.

8.1.11 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 8.1.3:

```
(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 several short-form sugar functions that recursively conform an instance to full-form.

8.1.12 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 sugar
        (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})
```

8.1.13 Term-Entity Keys

The actual `enum`-like macro also defines the *term-entity-key spec* for any `enum`-like (see the companion document, `masr-summary.pdf`)

```
(s/def ~tkw    ;; like ::intent
  (s/and ~art ;; like ::asr-term, i.e., the multi-spec
    ;; like the predicate #(= ::intent (::term %))
    (term-selector-spec ~tkw)))
```

In this case, the term-entity-key spec is `::intent`:

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

8.2 **TODO** unit

8.3 TODO symbol

8.3.1 Variable

Here is an example of the full form for a `::Variable` with a conforming instance in light sugar (see the companion document, `masr-summary.pdf`). Note the term-head-entity-key spec (see the companion document, `masr-summary.pdf`), `::Variable`. Any `::Variable` is also an `::asr-term`.

```
(let [a-var-head
      {::symbol-head ::Variable
       ::syntab-id    (nat 2)
       ::varnym       (varnym 'x)
       ::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       (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))
```

Here is an example in heavy sugar:

```
(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)
```

The source file, `specs.clj`, tests each of the 13 positional arguments of the heavy-sugar function `Variable--` for recursive conformance.

A macro, `Variable`, for compatibility with legacy ASDL, is provided. See the companion document, `masr-summary.pdf`, for more info on such compatibility.

8.4 storage_type

Storage-type is another enum-like, defined and registered via macro (Section 8.1.12). The following tests of full-form and heavy sugar illustrate conformance to both `::asr-term` and the term-entity-key spec, (see the companion document, `masr-summary.pdf`), `::storage-type`:

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

8.5 access

Access is another enum-like, defined and registered via macro (Section 8.1.12). The following tests of heavy sugar illustrate conformance to both `::asr-term` and the term-entity-key spec (see the companion document, `masr-summary.pdf`), `::access`.

```
(enum-like access #{'Public 'Private})

(tests
  (let [public (access 'Public)]
    (s/conform ::asr-term public) := public
    (s/conform ::access public) := public)
  (access 'foobar) := ::invalid-access)
```

8.6 TODO deftype

8.7 presence

Presence is another enum-like, defined and registered via macro (Section 8.1.12). The following tests of heavy sugar illustrate conformance to both `::asr-term` and the term-entity-key spec (see the companion document, `masr-summary.pdf`), `::presence`.

```
(enum-like presence #{'Required 'Optional})

(tests
  (let [required (presence 'Required)]
    (s/conform ::asr-term required) := required
    (s/conform ::presence required) := required)
  (presence 'fubar) := ::invalid-presence)
```

8.8 abi

Abi is a rich case. It is enum-like, similar to *intent* (Section 8.1), but with restrictions. Its heads include several *external-abis*:

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

and one *internal-abi*:

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

The *abi-enum* contents spec for *abi* is the union 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 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 in full-form:

```
(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 Interactive, :abi-external true}))
```

The sugar for *abi* is an exceptional case. We deem it better to default the `:external` Boolean to `false` in all cases except that for `'Source`, and to require an explicit `:external` keyword in other cases. That means that arity-1 usages like

```
(abi 'Source)
```

and arity-3 usages like

```
(abi 'Source :external false)
```

are valid, but arity-2 usages like

```
(abi 'Source false)
```

are not valid.

The following sugar function effects this design:

```
(defn abi
  ;; arity 1 --- default "external"
  ([the-enum]
   (let [abi_ (s/conform
                 ::asr-term
                 {::term      ::abi,
                  ::abi-enum  the-enum,
                  ::abi-external
                    (not (= the-enum 'Source))})]
     (if (s/invalid? abi_)
         ::invalid-abi
         abi_)))
  ;; arity 2 --- invalid
  ([the-enum, crap]
   ::invalid-abi)
  ;; arity 3 --- light sugar
  ([the-enum, ext-kw, the-bool]
   (cond
    (not (= ext-kw :external)) ::invalid-abi
    :else
     (let [abi_ (s/conform
                   ::asr-term
                   {::term      ::abi,
                    ::abi-enum  the-enum,
                    ::abi-external the-bool})]
       (if (s/invalid? abi_)
           ::invalid-abi
           abi_))))))
```

Here is its term-entity key (see the companion document, `masr-summary.pdf`), `::abi`, for recursive checking:

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

Here are some conformance tests for full-form, sugar, against `::asr-term`, and against the term-entity key `::abi`:

```
(tests
 (s/valid? ::asr-term
   {::term      ::abi
    ::abi-enum 'Source
    ::abi-external false})      := true
 (let [abe (abi 'Source :external false)]
   (s/conform ::abi      abe)      := abe
   (s/conform ::asr-term abe)      := abe
   ;; defaults to correct value
   (abi 'Source)                  := abe
   ;; missing keyword
   (abi 'Source false)            := ::invalid-abi
   ;; wrong value
   (abi 'Source :external true)   := ::invalid-abi)
 (let [abe (abi 'LFortranModule :external true)]
   (s/conform ::asr-term abe)      := abe
   (s/conform ::abi      abe)      := abe
   ;; defaults to correct value
   (abi 'LFortranModule)          := abe
   ;; missing keyword
   (abi 'LFortranModule true)     := ::invalid-abi
   ;; wrong value
   (abi 'LFortranModule :external false) := ::invalid-abi))
```

8.9 **TODO** stmt

8.10 **TODO** expr

8.11 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]))
```

8.11.1 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]]}})
```

8.11.2 Sugar for Integer, Real, Complex, Logical

Sugar for ttypes comes in two varieties, *light sugar* and *heavy sugar* (see the companion document, [masr-summary.pdf](#)).

1. Light Sugar Examples

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

2. Heavy Sugar Examples

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

See the tests for many more examples.

8.11.3 TODO Character

8.12 TODO restriction_arg

8.13 **TODO** binop

8.14 **TODO** logicalbinop

8.15 **TODO** cmpop

8.16 **TODO** integerboz

8.17 **TODO** arraybound

8.18 **TODO** arraystorage

8.19 **TODO** cast_kind

8.20 dimension

A *dimension* [*singular*] 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]])` (8.11), the one dimension [*singular*] in the dimensions [*plural*] (7.1) 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 [[]])` (8.11). This meaning of this type is an open question.²²

8.20.1 Empty Dimensions

Empty dimensions [*plural*], as in `(Integer 4 [])` specify non-array types. These are often called, loosely, *scalars*. Pedantically, *scalars* pertain only 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 7²² and in Section 7.1.

8.20.2 TODO: Issue 7: Zero Length

The following specs, in context of a ttype (8.11) for convenience, are legal in the ASDL grammar.⁹ 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]])
```

8.20.3 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)
```

8.20.4 Sugar

The following tests illustrate the sugar and the term-entity-key spec (see the companion document, `masr-summary.pdf`) 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

  (s/valid? ::dimension (dimension 60))          := false
  (s/valid? ::dimension (dimension []))          := false
  (s/valid? ::dimension (dimension 'foobar'))    := false
  (s/valid? ::dimension (dimension ['foobar']))  := false
  (s/valid? ::dimension (dimension []))          := true
  (s/valid? ::dimension (dimension [60]))        := false
  (s/valid? ::dimension (dimension [0]))         := false
  (s/valid? ::dimension (dimension '(1 60)))     := true
  (s/valid? ::dimension (dimension '()))         := true )
```

8.21 **TODO** alloc_arg

8.22 **TODO** attribute

8.23 **TODO** attribute_arg

8.24 **TODO** call_arg

8.25 **TODO** tbind

8.26 **TODO** array_index

8.27 **TODO** do_loop_head

8.28 **TODO** case_stmt

8.29 **TODO** type_stmt

8.30 **TODO** enumtype

9 Implicit Terms

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

Some items specified in ASDL as *symbol_table* are actually *syntab_id*.

9.1 syntab_id

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

```
(tests
  (syntab-id -42)           := ::invalid-syntab-id
  (syntab-id 'foo)          := ::invalid-syntab-id
  (syntab-id 42)            := 42
  (s/conform ::nat 42)       := 42
  (s/conform ::nat (nat 42)) := 42
  (s/conform ::syntab-id 42) := 42
  (s/conform ::syntab-id (syntab-id 42)) := 42
  (s/conform ::syntab-id (nat 42)) := 42)
```

9.2 TODO symbol_table

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