MASR — Meta Abstract Semantics Representation

Brian Beckman

10 Apr 2023

Contents

1 Abstract

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

Being agnostic means that it is easy to write new compilers, both at the front end and the back end. For example, LFortran predates LPython. When the need for a Python compiler arose, only a Python front end was necessary. Within a few days, a new end-to-end compiler, LPython, was created.

LCompiler back ends are completely reusable because ASR eliminates all original language syntax from the IR, in sharp contrast to typical practice, which treat semantics as decorations on syntax trees.

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

Current specifications for ASR are written in ASDL,[6] a metalanguage similar in spirit to yacc but less rich, by design.[7] To build an LCompiler like LFortran or LPython, the ASDL grammar[8] for ASR is parsed and a library, libasr,[9] in C++ is generated. Compiler front ends call this library to transform and emit ASR trees.

ASDL has several deficiencies, and MASR, [10] described in this document, alleviates them. We aim to replace libasr with MASR.

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 libasr. The document mirrors an ASDL snapshot.[8]

[1] `https://en.wikipedia.org/wiki/Intermediate_representation`
[2] `https://github.com/lcompilers/libasr`
[3] `https://lfortran.org/`
[4] `https://lpython.org/`
[5] `https://webassembly.org/`
[6] `https://en.wikipedia.org/wiki/Abstract-Type_and_Scheme-Definition_Language`
[7] `https://en.wikipedia.org/wiki/Yacc`
[8] `https://github.com/rebcabin/masr/blob/main/ASR_2023_APR_06_snapshot.asdl`
[9] `https://github.com/lfortran/lfortran/tree/c648a8d824242b676512a038bf2257f3b28dad3b/src/libasr`
[10] pronounced "maser;" it is a Physics pun

2 Issues with ASDL

2.1 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.

2.2 ASDL is Incomplete

Much of the semantics of ASR is currently 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.

2.3 ASDL's ASR is Volatile

The ASDL for ASR changes frequently, for good reasons. However, stand-aside tools like asr-tester[11] 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.

2.4 ASDL is Ambiguous

There are many syntactic and semantic ambiguities in the ASDL grammar.[8]. For example, the type notation `integer*` might mean, in one place in the grammar, a list of `integer` with duplicate entries allowed, and, in another place in the grammar, 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 the 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.

[11] `https://github.com/rebcabin/asr-tester`

3 Clojure Solves ASDL Issues

ASR expressions, being trees, have a natural representation in S-Expressions.[12] Clojure, being a modern Lisp, natively handles S-Expressions. Clojure is modern. Clojure has a robust, lively ecosystem.

Clojure.spec,[13] 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 2.4. 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.[14]. 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.

[12] `https://en.wikipedia.org/wiki/S-expression`
[13] `https://clojuredocs.org/clojure.spec.alpha`
[14] `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.[8] Names of terms are generally in lower-case.

Table 1 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 `symtab_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 2.4. 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 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 footnote[15] for an informal description of *PascalCase*.

There are of 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.

[15] https://alok-verma6597.medium.com/case-styles-in-development-camel-pascal-snake-and-kebab-case-ed8

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

Entity Hash-Maps — ASR entities[16] shall be hash-maps with fully-qualified keywords as keys (see Section 8.1 for motivating example).

Multi-Specs — ASR entity hash-maps shall be recursively checked and generated via Clojure multi-specs.[16]

Explicit — ASR entity hash-maps shall contain all necessary information, even at the cost of verbosity. Defaults are not permitted.

Syntax Sugar — Extra constructor functions for ASR entities may allow default values for positional and keyword arguments. See Section 8.11 for an example and see Issue 3 on MASR's GitHub repo.[17]

[16] `https://clojure.org/guides/spec`
[17] `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?)
```

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 0x7FFFFFFFFFFFFFFF))               := true)
```

6.2 Notes

A Clojure *int* is a Java *Long*, with some peculiar behavior for hex literals.[18] 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 in MASR. To access the underlying `java.lang.Long`, one must employ Clojure's `unchecked-long`.

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

[18] `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`, `identifier`, `identifiers`. Note carefully the singulars and plurals in the names of the specs.

7.1 dimensions [*sic*]

7.2 identifier [*sic*]

7.3 identifiers [*sic*]

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.[16]

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.[19]

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.

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.[13] Therefore the spec for *intent contents* is just the set of valid members.

[19] `https://docs.cider.mx/cider/debugging/debugger.html`

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*,[16] `::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-maps[20] 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`[21] — 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.[22]

`defmethod`[23] — 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.[24]

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:

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

[20] https://clojuredocs.org/clojure.core/hash-map
[21] https://clojuredocs.org/clojure.core/defmulti
[22] https://stackoverflow.com/questions/6915531
[23] https://clojuredocs.org/clojure.core/defmethod
[24] Multi-specs allow re-tagging, but we do not need that level of generality.

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`.

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.

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 `::term` keyword and an `::intent-enum` keyword.

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 mult-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 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 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 a short-form sugar function that does a conformance check.

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 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})
```

8.2 unit

8.3 symbol

8.3.1 TODO Variable

8.4 storage_type

8.5 access

8.6 deftype

8.7 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, 'Intrisic})
```

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

8.8.1 Syntax Sugar

The sugar for *abi* uses Clojure destructuring[25, 26] 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)
```

[25] https://clojure.org/guides/destructuring
[26] https://gist.github.com/rebcabin/a3c24be3e17135f355348c834ab14141

Non-conforming due to incorrect types or structure:

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

8.9 stmt

8.10 expr

8.11 ttype

Ttype [*sic*] has 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] [82]]}})
```

8.11.2 Sugar for Integer, Real, Complex, Logical

Sugar for ttypes comes in two varieties, *light sugar* and *full sugar*. Light sugar require specs with keywords, as in:

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

Full sugar uses positional arguments, as in

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

See the tests for many examples.

8.11.3 TODO Character

8.12 restriction_arg

8.13 binop

8.14 logicalbinop

8.15 cmpop

8.16 integerboz

8.17 arraybound

8.18 arraystorage

8.19 cast_kind

8.20 dimension

A *dimension* is 0, 1, or 2 nats in a Clojure list or vector:

```
(def MIN-DIMENSION-COUNT 0)
(def MAX-DIMENSION-COUNT 2)
(s/def ::dimension-content
  (s/coll-of ::nat
             :min-count MIN-DIMENSION-COUNT,
             :max-count MAX-DIMENSION-COUNT,
             :into ()))
```

If there is one nat, it specifies the length of any array dimension that enjoys the instance. For example, in the ttype `(Integer 4 [[42]])` (8.11) , the one dimension in the dimensions [*sic*] (7.1) of the ttype is `[42]`, and the ttype specifies a rank-1 array of 42 4-byte integers, with indices starting at 1 and running through 42.

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 in the dimensions [*sic*] (7.1) of the ttype is `[6 60]`, and 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 **unspecified**. For an example, consider the ttype `(Integer 4 [[]])` (8.11).

8.20.1 TODO: Issue 6: Empty Dimension

See Issue 6 in MASR's GitHub repo[27] for discussion of the meaning of `(Integer 4 [[]])`. Empty dimensions [*sic*] as in `(Integer 4 [])`, are discussed 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.[8] The meaning is **unspecified**:

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

See Issue 7 in MASR's GitHub repo.[28]

8.20.3 TODO: Issue 5: Unspecified Starting Index

If there is only 1 nat, the starting index is unspecified. In most programming languages, the default is 0. In Fortran and Mathematica (Wolframscript[29]), the default

[27] `https://github.com/rebcabin/masr/issues/6`
[28] `https://github.com/rebcabin/masr/issues/7`
[29] `https://www.wolfram.com/wolframscript/`

is 1. We provisionally adopt that convention while the issue is open. See Issue 5 in MASR's GitHub repo.[30]

8.20.4 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]})    := true
 (s/valid? ::asr-term
           {::term ::dimension
            ::dimension-content []})     := true)
```

8.20.5 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 [60]))    := true
 (s/valid? ::asr-term (dimension [0]))     := true
 (s/valid? ::asr-term (dimension []))      := true
 (s/valid? ::asr-term (dimension '(1 60))) := true
 (s/valid? ::asr-term (dimension '()))     := true)
```

[30]https://github.com/rebcabin/masr/issues/5

8.21 alloc_arg

8.22 attribute

8.23 attribute_arg

8.24 call_arg

8.25 tbind

8.26 array_index

8.27 do_loop_head

8.28 case_stmt

8.29 type_stmt

8.30 enumtype

8.31 Implicit Terms

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

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

8.31.1 symtab_id

8.31.2 symbol_table

9 Change Log

2023-06-Apr :: Start.

2023-12-Apr :: enum-like specs