Use (and abuse?) of Ada 2022 features in designing a JSON-like data structure
Is this valid Ada?

Var : String_Array := ("one", "two", "six");
type String_Array is array (Positive range <>) of String (1 .. 3);  ✓

Var : String_Array := ("one", "two", "three");
type String_Array is array (Positive range <>) of String;  ✗

Var : String_Array := (+"one", +"two", +"three");
type String_Array is array (Positive range <>) of Unbounded_String;  ✓
function +" (Str : String) return Unbounded_String;
Var : Yeison.Str := “Hello, FOSDEM!”; type Str is new String; ✓

type Str is new Any with private; ⁉

Ada 2012  ❌
Ada 2022 (-gnat2020, 11.2.3) ✓

Var : Str_Array := (“two”, “three”); type Str_Array is array of Str; ✓ -- (but how?)

type Str_Array is tagged private; ⁉
package Yeison is

  type Str is ... with String_Literal => To_Str;

  function to_Str (S : Wide_Wide_String) return Str;

  type Str_Array is ... with Aggregate => (Add_Unnamed => Append);

  procedure Append (Container : in out Str_Array; Element : Str);

end Yeison;
Maps

V : Yeison.Vec := ("two", "three"); ✔

M : Yeison.Map := ("one" => 1,
                  "two" => 2);

type Map is … ✔
  with Aggregate =>
    (Add_Named => Insert);

procedure Insert (Container : in out Map; ✔
                 Key       : Key_Type;
                 Value     : Element_Type);
JSON, YAML, TOML, Ada

```ada
X : Yeison.Map :=
    ("array" => (1, "2", 3.0),
     "boolean" => True,
     "color" => "gold",
     "number" => 123,
     "object" => ("a" => "b",
                 "c" => "d"));
```

```json
{
    "array" : [1, "2", 3.0],
    "boolean" : true,
    "color" : "gold",
    "number" : 123,
    "object" : {
        "a": "b",
        "c": "d"
    }
}
```

array = [ 1, "2", 3.0 ]
boolean = true
color = "gold"
number = 123

X : Yeison.Map :=
    ("array" => (1, "2", 3.0),
     "boolean" => True,
     "color" => "gold",
     "number" => 123,
     "object" => ("a" => "b",
                 "c" => "d"));
```

```ada
array:
- 1
- ‘2’
- 3.0

boolean: true

color: gold

number: 123

object:
    a: b
c: d
```
type Str is ...
  with String_Literal => To_Str;

function to_Str (S : Wide_Wide_String) return Str;

type Big_Int is ...
  with Integer_Literal => To_Big_Int;

function To_Big_Int (S : String) return Big_Int;

X : Big_Int := 123456789012345678912345678912345678912345678;
1st outlandish idea

What if...

type Any is private with
  Integer_Literal => To_Any,
  Real_Literal    => To_Any,
  String_Literal  => To_Any;

$ alr build
Building yeison/yellowoan.gpr...
Build finished successfully in 3.12 seconds.

N : Yeison.Any := 1;
R : Yeison.Any := 3.14;
S : Yeison.Any := "What could possibly go wrong";
2nd outlandish idea

So then, what if...

type Collection is private with
    Aggregate => (Add_Named => Insert, ??
                    Add_Unnamed => Append); ??

$ alr build
Building yeison/yeison.gpr...
yeison.ads:73:19:
conflicting operations for aggregate (RM 4.3.5)

ARM 202X 4.3.5 (5/5): If Add_Named is specified, neither Add_Unnamed nor Assign_Indexed shall be specified.
So, you need at least two types, one for maps or vectors exclusively.
Using single type + auxiliary type

1st serious attempt:

type Any is tagged private with
  Integer_Literal   => To_Int,
  Real_Literal      => To_Real,
  String_Literal    => To_String,
  Aggregate => (AddNamed => Insert);

type Vec_Aux is private with
  Aggregate => (AddUnnamed => Append);

function Vec (V : Vec_Aux) return Any;

X : Yeison.Map :=
  ("array"   => Yeison.Vec ((1, "2", 3.0)),
   "boolean" => Yeison.True,
   "color"   => "gold",
   "number"  => 123,
   "object"  => ("a" => "b",
                  "c" => "d");
Instead of a single type, rely on a single type class:

```ada
type Any is tagged private;

type Map is new Any with
    Aggregate => (Add_Named => Insert);

type Vec is new Any with
    Aggregate => (Add_Unnamed => Append);

procedure Append (Container : in out Vec;
    Value     :   Any’Class);

type Int  is new Any with private with Integer_Literal => To_Int;
type Bool is new Any with private;  function True return Bool;
type Real is new Any with private with Real_Literal    => To_Real;
type Str  is new Any with private with String_Literal  => To_Str;
```

Can we do better?
I : Yeison.Int := 1; ✓
S : Yeison.Str := “SNAFU”; ✓
V1 : Yeison.Vec := (I, S); ✓

V2 : Yeison.Vec := (1, “two”); ⚫
expected type "Any'Class" defined at yeison.ads:10
demo.adb:24:55: found a string type

V2 : Yeison.Vec :=
    (Yeison.Int'(1),
     Yeison.Str'("two")); ✓
type Any is tagged private with
    Integer_Literal => To.Any,
    Real_Literal    => To.Any,
    String_Literal  => To.Any;

V : Yeison.Vec := (1, "two"); ✓
M : Yeison.Map := ("one" => 1,
    "two" => 2.0,
    "three" => V); ✓

V2 : Yeison.Vec := (1, "two", (3.0, "four")); ?

demo.adb:49:42: type of aggregate cannot be class-wide
demo.adb:49:42: type of aggregate cannot be class-wide

\[(3.0, "four")\]  -- Can only be a Yeison.Vec

\[V2 : Yeison.Vec := (1, "two", \ldots \);\]  -- Which is Any’Class!

\[V2 : Yeison.Any’Class := (1, "two");\]  -- Same error

\[V2 : Yeison.Any’Class := Yeison.Vec’(1, "two");\]  ✓

Any lawyer in the room?
Not all stories have happy endings

X : Yeison.Map :=
("array" => Yeison.Vec'(1, "2", 3.0),
  "boolean" => Yeison.True,
  "color" => "gold",
  "number" => 123,
  "object" => Yeison.Map'("a" => "b",
     "c" => "d"));
type Any is tagged private with
  Constant_Indexing => Constant_Reference,
  Integer_Literal    => To_Int,
  Real_Literal       => To_Real,
  String_Literal     => To_Str;

type Int is new Any with private with
  Integer_Literal => To_Int;

function False return Bool;
function True  return Bool;

type Map is new Any with private with
  Aggregate => (Empty => Empty,
                 Add_Named => Insert);

type Vec is new Any with private with
  Aggregate => (Empty => Empty,
                 AddUnnamed => Append);
• Remove (arbitrary?) restriction from 4.3.5
• Allow aspect overloading:

```ada
type Any is tagged private with Constant_Indexing => Constant_Reference;

function Constant_Reference (This : Any'Class; Pos : Positive) return access constant Any'Class; ✓
function Constant_Reference (This : Any'Class; Key : String) return access constant Any'Class; ✓
function Constant_Reference (This : Any'Class; Path : Vec'Class) return access constant Any'Class; ✓

-- Allowed to be able to index by index/key and cursor

-- Ignored for aggregate initialization

procedure Append (Container : in out Vec; Value : Map'Class);
procedure Append (Container : in out Vec; Value : Vec'Class);
```

-- Only last one is recognized
M : Yeison.Map :=
    ("one" => Yeison.Vec("Hello", "FOSDEM"));

What’s at ("one", 2)?
-- FOSDEM

M ("one")(2) -- with Constant_Indexing => …

Those that lurk in the shadows…
M (My_Vec)
M (("one", 2))

demo.adb:99:16: ambiguous expression (cannot resolve "Constant_Reference")
demo.adb:99:16: possible interpretation at yeison_classwide.ads:24
demo.adb:99:16: possible interpretation at yeison_classwide.ads:19

M (Yeison'Vec ("one", 2))

function "/" (L, R : Any’Class) return Vec; -- Path-like
M ("one" / 2)
But WHY??

• We do what we must  
  Because we can

• Write config files in Ada  
  Verify with the compiler  
  Parse during run-time

  … as Alire did once upon a time.
Conclusion

- User-defined literals strike a good balance between
  - implicit conversions (C++)
  - explicit misuse (“+”)
- Aggregate initialization builds nicely on top of user-defined literals
  - Elements can be user-defined literals
  - Unfortunate (?) restriction deprives of some flexibility
    - There may be alternatives down the road
Thanks for your attention