Yafl == YAFL == Yet Another Func(tional) Lang(uage)
is a *limited* abstract pure function type specification intended for limited data, function, and type compatibility among functional programming environments including Scala, Haskell, Idris, Agda, Lean and F*. The full name of the current spec is YaflSpec v0.1.0.

YaflSpec is defined in layers corresponding to universes of (intensional, Martin Lof) types.
Yafl defines a specification of datatypes and pure functions derived from typed lambda calculus, with certain restrictions defined at each specification layer, corresponding to a type universe.
Particular Yafl functions may be proven decidable, but in general Yafl functions may not be assumed to be decidable. Particular functions may be proven decidable or undecidable.
Some implementation environments may require that only provably decidable functions may be executed, and may enforce other restrictions on the expected size of computation.
In experimental environments, some functions might be evaluated undecidably and with unknown size.

YaflSpec does not specify any runtime requirements.

YaflSpec v0.1 outlines four layers of specification, in alphabetical order of inclusion

YaflSpec-Core (YaflCore) defines a model of a limited space of algebraic datatypes, with limited induction (restricted self-inclusion). YaflCore includes a single category of one-arg pure function types which may only process ordinary data, and may not explicitly refer to functions or types as data. (No lambda arguments, no closures, …).

YaflSpec-Full (YaflFull) defines a limited model of pure functions with arguments and results drawn from (YaflCore | YaflFull) including limited use of functions and types as explicit data items.

YaflSpec-More (YaflMore) defines additional useful but less-common datatypes including binary data and data artifacts designed to support execution. Functions using YaflMore types are more limited, and may not be treated as first-class data themselves.

YaflSpec-Proof (YaflProof) allows for specification of proofs regarding types in the lower layers, to strengthen our expectations about Yafl functions and types.

By abuse of notation we may now say
YaflSpec = YaflCore | YaflFull | YaflMore | YaflProof

To use YaflSpec directly requires implementation of (some part of) the YaflSpec type system and deployment to some runtime such as JVM, LLVM, WASM

YaflCore and most of YaflFull may be implemented easily in Haskell or Scala.
YaflProof (+ Full and Core) should be completely implementable in hybrid proof+execution systems like Idris, F-Star, Lean 4.

YaflMore is a placeholder of relatively low priority in our specification process as of 2022-Jan.

YaflSpec Core Data is intended to be 100% compatible with JSON as an exchange format. Any strongly typed CoreData may be written and read as JSON, and any (finite) JSON input may be treated as weakly typed Core Data.

A weak version of YaflFunc functions may be used in loosely typed environments like JavaScript.

YaflCore datatypes may use a limited form of induction.
YaflCore functions

is defined in theory as a space of pure typed 1-arg functions, where arg + result types have a common type bound. In scala we may sketch the primary Yafl function type like so, but note we get no guarantee of purity from this trait.

Implementor must supply a pureEnoughCalc which is sufficiently trustworthy for system using Yafl. // This allows arbitrary system integration with all the attendant risks (and sometimes practical benefits).

Note that by using explicit functions-as-data in a Yafl-aligned language like axLam, functions are // constructed which are known to be pure, and are subject to further proofs (although mistakes in the space of "known" and "proven" are also possible).

Here we focus on Yafl-Core which supports composition of functions over strong type of immutable data.
// YaflCore is defined as functions like YaflPureFunc01[YaflCoreDat]
All YaflCoreDat is immutable

grounded in YaflPrimDat types like YaflNum and YaflTxt which are all primitive subtypes of YCoreDat.

The composite subtypes of YCoreDat are derived from these a limited , roughly
 1) Records/Cartesian-Products (N-tuples of input types)
 2) Sums of alternatives, like Either
 3) List
axLam : Yafl
axLam is an YaflFunc instance triangle: RDF, JSON, Scala

Limited spaces of YaflFuncs are constructed as axLam func descriptions
as subtypes of YaflFunc01 which are one-arg functions.
Multi arg functions YaflFunc02... are sugar-only in our axLam impl.

Actual impls of AxLam funcs are of two kinds:

1) Authored functions describing typed Lambda functions of Ydat
2) Coded runtime funcs implementing equivalent of YaflFunc01 (Java/Scala or LLVM or ...) in a way that is "good enough"
for the mission. Goodness of the functions is the subject of proofs.

Here is a taxonomy of YaflCoreDat types as scala traits without type parameters nor methods.
These are design markers aligned to YaflCore specification.
 (As of 22-01-03, these *are* the only specifcation)
An instance of YaflCoreDat is an item of YaflCore data.
Note that YaflCoreDat does not directly support URIs at this level.
Instead URIs may be defined (constructively?) in a specific more limited,
higher level authoring language, e.g. axLam

YaflCore offers 3 composite types : Cartesian Product, Alternative Sum, Finite List

YaflCartProd = Dependent-typed cartesian product, equivalent to Tuple-N over types T1, T2 .. TN

YaflAltSum = Dependent-typed sum, equivalent to some N-chain of Eithers over types T1, T2 .. TN

Above two types are sufficient for algebraic composition purposes of what we may call finite intensional types.

As a design choice we include one further dependent type, Finite list.
YaflFinList is a dependent-typed finite list of items of known length (equivalent to a Product)
where 1) all items come from some fixed type T <: YaflCoreDat
and 2) yaflList Len (a number term) is included in type of the FinList (see Barendregt lambda cube)

Clarifying the "redundancy": Algebraically the YaflFinList type is equivalent to a Sum over different length Products

https://datatracker.ietf.org/doc/html/rfc7159#page-5

3. Values

A JSON value MUST be an object, array, number, or string, or one of the following three literal names:

false null true

The literal names MUST be lowercase. No other literal names are allowed.

```
value = false / null / true / object / array / number / string

false = %x66.61.6c.73.65   ; false

null  = %x6e.75.6c.6c      ; null

true  = %x74.72.75.65      ; true
```