JSON library for LLVM (draft)

sammccall@google.com, 2017-10-20. This is a PUBLIC doc, audience is the LLVM community.
It's ordered from platitudes to implementation details: I hope to lead you from agreement to bikeshedding :-)

The problem

The LLVM project has various components that produce or consume JSON - clangd in particular, because it is a JSON-RPC server. The way this is done has problems:
Producers mostly use printf (or moral equivalent), with yaml::escape for strings.

- it's easy to get syntax wrong (e.g. quoting and escaping, trailing comma)
- generating code is hard to read/format, especially when it needs to be efficient
- complex or arbitrary JSON objects must be passed as strings, weakening contracts
- lit tests depend on the exact ordering of fields, which is an implementation detail

Consumers use YAMLParser, taking advantage of the fact that YAML is a superset of JSON.

- they necessarily accept invalid JSON documents
- code needs to use unfamiliar YAML terminology and handle YAML-only conditions (e.g. non-string keys)
- the streaming parser is hard to use, e.g. no random-access to object properties. YamlIO has its own limitations on data structures, and can't currently produce JSON output.
- the streaming parser fundamentally can't handle tagged-union JSON objects where the tag may appear later in the stream than the other data. This occurs in the Language Server Protocol implemented by clangd, and today clangd is incompatible with some editors for this reason. YamlIO does not solve this.

Proposal

We should write a new DOM-oriented library for JSON parsing, serializing, and manipulation.
By DOM-oriented, I mean you can parse or construct a Document object: an in-memory tree of JSON values that supports random-access, editing, and can be serialized into JSON.

Alternatives:

- **The status quo** is frustrating for the reasons above.
- **Add a JSON mode to YAML** which would only produce/consume JSON. Yaml/YamlIO could be used for producing too, which makes defining mappings more worthwhile. I think this would still produce a confusing API for JSON users, and makes the implementation more complex. It doesn't solve the tagged-union problem, which would be pretty painful for clangd.
  YamlIO is not to my taste:
  - it imposes on your class, requiring mutation rather than construction makes invariants less idiomatic
  - code is "more frameworky" than imperative mapping - behavior is standardized, but needs a lot of context to understand
  - simple cases are simple, but nontrivial cases require special idioms (e.g. normalization). Not clear to me it saves code overall

  Still, this is quite a strong alternative. It fixes most of the problems I'm complaining about, and should have better performance than a DOM if we're eventually going to convert to domain objects.

- **Import a third-party JSON library** (discussion)**.** There are lots to choose from, but using third-party code is problematic in some parts of LLVM, and uniformity is valuable. Third-party libraries will be less well integrated with LLVM facilities (allocators, raw_ostream, ...). Modifying them to fit our needs is technically possible, but maintaining a fork loses much of the benefit of using existing code.
- **Write a streaming JSON library** like YAMLParser. This is hard to use directly for common things, and probably necessitates something like YamlIO. The conceptual overhead of two YamlIO-like things is high, and I don't know how to make it substantially better. So I think adding a JSON mode to YAML dominates this option.

Major design choices/goals

This should be a useful general-purpose library for the LLVM project. Efficiency is important but not paramount.

Parsing:

- parse the whole document upfront, and do not attempt error recovery - just report an error and fail. This leads to less surface area where errors need to be handled.
- we should avoid lots of tiny mallocs while parsing, the whole document should share a slab allocator

Serialization:

- serializing in canonical form (prettyprinted, sorted keys) should be possible (useful for tests)

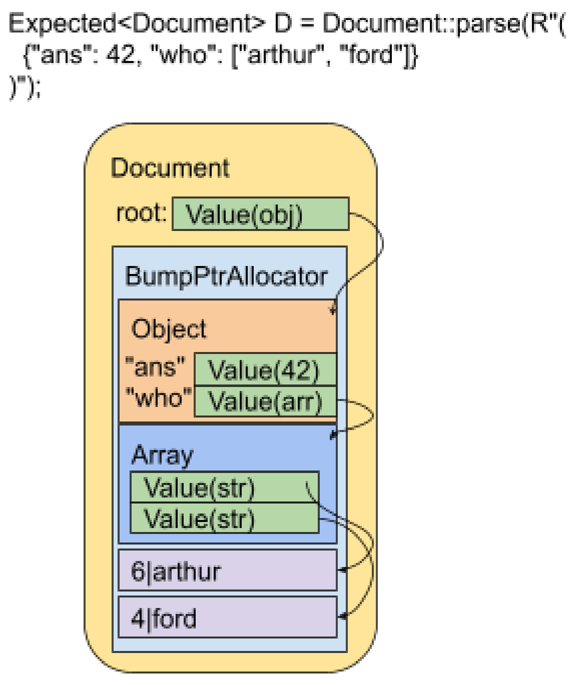
DOM representation:

- compose objects with a natural literal syntax, e.g.
```
Document D = json::obj{
                        {"foo", 42},
                        {"bar", {"a", nullptr, SomeJSONObject }},
                };
```
  (there's **lots** of subtle details to get right here - probably the hardest part).
- references to parts of a document should be possible, but copies/moves need not be cheap

Object mapping:

- Don't provide opinionated "framework-like" support, but make it natural to express JSON tree → T and T → JSON tree transformations as functions that compose well (i.e. can build from the inside out).

Data structures



```
Expected<Document> D = Document::parse(R"(
  {"ans": 42, "who": ["arthur", "ford"]}
)");
```

json::Document { BumpPtrAllocator, Value }

This is a self-contained JSON value. Its allocator owns the whole tree, and it is movable.
A Document also exposes the Value API (acting as its own root).

json::Value { PointerSumType<...> }

This is a generic JSON value owned by a Document. Users only get references to these.
It points to a representation in the doc's allocator (Object, Array, double, pascal-string). Nulls, booleans, and small integers are stored inline.

The API is that of a discriminated union:
  Optional<StringRef> string() const
  Array* array()
etc.
You can't assign to an object (it doesn't know where to allocate memory), you mutate to its container instead.
A MyType::fromJson() function would accept a Value.

json::Array { vector<Value, BumpPtrAllocator&> }

A mutable array JSON value owned by a Doc. Users only get references.
This supports a vector-like API. Wrinkle: you read Value&s, but you write Exprs.

json::Object { StringMap<Value, BumpPtrAllocator&> }

A mutable array JSON value owned by a Doc. Users only get references.
This supports a map-like API. Similar to Array, reading and writing are different types.

json::Expr {...}

This is a JSON structure not owned by a document.
It's generally created by a literal expression like `return {1, true, json::obj{{"foo", buildFoo()}}};`
These are opaque and immutable, their main purpose is to be able to be inserted into Documents or composed into other Exprs from the inside-out. A MyType.toJson() method would return an Expr.

Prototype

There's a prototype in https://reviews.llvm.org/D39180 with more details fleshed out.
Some things aren't right:

- Expr is called Literal
- StringMap and vector can probably be replaced with better specialized types.
  (In particular, large JSON documents often have many objects with the same string keys. These could be shared)