tree-gen Documentation

QuTech, TU Delft

Sep 22, 2023

tree-gen

tree-gen is a program that generates C++ source code for tree structures based on a simple and concise input language. The API documentation is generated by Doxygen.

CHAPTER 1

Overview

tree-gen is a code generator that outputs all the repetitive stuff needed for defining a proper tree structure in C++.

Trees are described by *.tree files. Such a file consists of a number of directives at the top and one or more tree node descriptions. Each tree node gets its own C++ class, with a bunch of methods defined on it for traversal, the visitor pattern, safe typecasting, cloning, and so on. It would be extremely tedious and error prone to define all this stuff manually, hence the generator.

It is important to realize and perhaps not immediately obvious that the recursive structure in a *.tree file represents C++ inheritance rather than the actual tree structure. You may for instance encounter something like this:

```
expression {
    addition {
        ...
    }
    subtraction {
        ...
    }
}
```

That doesn't mean an expression consists of an addition and a subtraction, but rather that an addition is a type of expression, and subtraction is another.

1.1 Base classes and primitives

tree-gen trees consist of three kinds of objects: nodes, edges, and primitives.

The nodes are classes generated by tree-gen based on some base class. This is typically tree::base::Base. The class inheritance tree of the node types is used for "union" types, i.e. to allow some node in the tree to be one of a number of types. For example, some node may want to have an expression child node, which in practice would be for instance an addition node or a subtraction node. At any level of this class hierarchy, nodes can be given attributes to store data. These attributes are either edges or primitives.

Edges define the relation between some parent node and some number of child nodes. Six edge classes are defined, resulting in different relations between the nodes.

- `tree::base::Maybe<T>`: the parent will own zero or one child node of (super)type T.
- `tree::base::One<T>`: the parent will logically own exactly one child node of (super)type T.
- `tree::base::Any<T>`: the parent will own zero or more child nodes of (super)type T.
- `tree::base::Many<T>`: the parent will logically own one or more child node of (super)type T.
- `tree::base::OptLink<T>`: the parent links to zero or one child node of (super)type T elsewhere in the tree.
- `tree::base::Link<T>`: the parent logically links to exactly one child node of (super)type T elsewhere in the tree.

The "logically" term used above alludes to the fact that One/Many/Link don't necessarily need to have at least one child at all times. They may for instance be empty during construction. But at least one child node is required for the tree to be considered to be "complete". Furthermore, completeness with respect to some root node requires that any Link or non-null OptLink refers to a node that is actually reachable from that root node.

Maybe and One are based on std::shared_ptr, Any and Many are based on std::vector<One>, and OptLink and Link are based on std::weak_ptr, but these types are abstracted away from the user almost entirely. Notably, all exposed dereference operations are null- and range-checked, throwing exceptions if there's a problem, rather than causing a segmentation fault down the line.

The namespace for the Base and edge classes can be set using the `tree_namespace` directive in case you want to override them or add functionality. Note that you can use "using One = tree::base::One;" etc. if you only want to override part of the classes. You can also opt to use the `tree-all.[ch]pp.inc` files, which define the same structures that exist in the `tree` namespace, but allows you to override some of the standard library types they are based on, for example to use a different base exception type. If the tree_namespace directive is not specified, the namespace that the node classes are generated into will be used by default.

Finally, primitives are used to store the actual data, forming the leaves of the tree. They can generally be any externally-defined C++ type, although a few template functions need to be defined and specialized for the types you want to use. These are:

```
template <typename T>
T initialize() { ... }

template <typename T>
void serialize(const T &obj, tree::cbor::MapWriter &map) { ... }

template <typename T>
T deserialize(const tree::cbor::MapReader &map) { ... }
```

The initialize function is used to construct a primitive of type T in such a way that it has a defined value. This is in contrast to the behavior of C's own primitive types (int, bool, etc.), which are specified to be undefined until assigned. This is avoided using this initialize function to prevent confusion caused by undefined behavior. Typically, initialize() can be implemented using just `return T{};` for the general case to defer to the constructor for complex types, and then be specialized for the primitive types that you want to use.

The serialize and deserialize functions are optional. When given, logic is generated to serialize and deserialize entire trees to and from a CBOR representation.

In addition to the type-safe primitives declared in the tree description file, nodes can also be annotated using arbitrary types. Specifically, every node can be annotated with zero or one annotation of every C++ type. So, any struct or class you make can be attached to any node in the tree, regardless of the tree specification. This is useful, for instance, to carry metadata, such as line number or debugging information. When tree-gen is used to generate some user-facing tree structure in a library, users of the library can also use this to add their own data to a tree as they operate on it, to prevent them from having to convert to their own data structure first.

Note that the above requires RTTI to be enabled and supported by the compiler, but there's really no good reason to disable that nowadays.

1.2 Defining node types

Nodes have the following form in the tree file:

```
# [documentation for node class]
[snake_case_node_name] {

    # [documentation for primitive]
    [snake_case_member_name]: [C++ namespace path];

    # [documentation for edge/child node]
    [snake_case_member_name]: [Maybe|One|Any|Many|OptLink|Link]<[snake_case_node_
↪name]>;

    # [documentation for external edge/child node]
    [snake_case_member_name]: external [Maybe|One|Any|Many|OptLink|Link]<[C++␣
↪namespace path]>;

    [zero or more specializations of this node; recursive structure]

    [optional reorder directive]
}
```

Nodes can have zero or more primitives, zero or more edges/child nodes, zero or more *external* edges/child nodes, and zero or more specializations.

The difference between normal edges and external edges is that the type for a normal edge must resolve to another node in the same tree file, while an external edge may use any C++ type name for the node class. That allows you to refer to nodes defined in other tree files as well. Note however that the generator expects external nodes to actually be node classes, such that it can properly generate the debug dump indentation logic, among other things.

Note that while node type names are specified in snake_case in the tree file, the name will be converted to TitleCase for the generated class names to follow decent C++ naming conventions. This is just because it's easier to convert snake case to title case than vice versa, and the generator uses both forms (title case for the class names, snake case for methods and members).

Note also the comments explicitly added to the example. Comments using a # are interpreted as docstrings by the generator; they are copied into the source code using javadoc-style comment blocks. This also means you can't use #-"comments" everywhere, since they're actually a grammatical construct. Therefore, tree-gen also supports // comments. These are completely ignored by the parser.

The order in which nodes are defined doesn't matter. This allows you to make recursive tree structures, without having to worry about forward declarations. The generator handles those. The order in which the attributes of a node are defined does matter, though, as it's used for the order in which the values can be passed to the node constructor; they are ordered subclass to superclass, top to bottom.

The order of the primitives and child notes (a.k.a. fields) is used in a few places, primarily the constructor and for debug dump output generation. The order is as specified within each class hierarchy level, but the fields of the more-specialized classes come before the fields of their ancestors.

When additions are made to a tree, it may be required to maintain compatibility with older tree versions by maintaining field order. In order to support this, the above field order can be overridden using a `reorder` directive. This directive has the following syntax:

```
# [optional documentation (ignored)]
reorder(a, b, c);
```

where `a, b, c` is the desired order of the fields. Any fields not explicitly specified in the list will automatically appear at the end, using the default order.

1.3 Directives

The following directives exist. They should be placed at the top of the tree file, before the first node. Their order doesn't matter, unless otherwise specified. Note that most directives are required.

- `source`: used to specify documentation for the generated source file; any docstring above the directive is copied into the file as file-level doxygen documentation.
- `header`: used to specify documentation for the generated header file; any docstring above the directive is copied into the file as file-level doxygen documentation.
- `python`: used to specify documentation for the generated Python file (if enabled via the command line); any docstring above the directive is copied into the Python file's docstring.
- `tree_namespace <namespace::path>`: the namespace that the Base and edge classes live in. You should set this to `tree::base` unless you intend to override or use different base classes. This is unused in Python.
- `initialize_function <namespace::path::initialize>`: the name (including namespace path leading up to it) of the `T()` function used for getting the default value of any of the used primitive classes. Unused in Python; here, the default constructor is used regardless of type (as it always exists and is meaningful in Python).
- `serdes_functions <namespace::path::serialize> <namespace::path::deserialize>`: optionally, the names (including namespace paths leading up to them) of the functions used to respectively serialize and deserialize primitive classes. If not specified, serialization functionality is disabled in C++. Python always has serdes support, but support is augmented when a serdes function pair is specified. The Python generator interprets the namespace path as a module path. Refer to the serialization section for more info.
- `location <namespace_path::SourceLocation>`: optionally, the name of the source location annotation class (including namespace path leading up to it). The debug dump will look for an annotation of this type on each node when doing a debug dump, and if it exists, uses it to add source information to the node, by streaming out a `#` followed by the stream overload for the class. The base class needs to be capable of annotations if you use this. This is not supported in Python.
- `include "<path>"`: adds an `#include` statement to the top of the generated C++ header file.
- `src_include "<path>"`: like `include`, but adds to the top of the generated C++ source file only.
- `import ...`, `from ... import ...`: adds Python import statements to the top of the generated module.
- `namespace <name>`: used to specify the namespace for the generated tree classes. As in C++, you need multiple of these to specify the full path. The docstring for the *first* annotation of this type that has a docstring in front is used to document the innermost namespace javadoc-style for Doxygen documentation. This is not used in Python.

1.4 Generated APIs

The following methods are generated for each node class:

- a constructor with all members of the node in its signature as optional values, defaulting to the values returned by the initializer function.
- `bool is_complete() const`: returns whether the node its called on and any subtree rooted in that node is fully defined (no empty One, Many or Link, all array entries in Any/Many nonempty, and all Links and defined OptLinks reachable from the root node).
- `%NodeType type() const`: returns the type of this node, using the also-generated %NodeType enumeration.
- `One<Node> copy() const`: returns a shallow copy of this node.
- `One<Node> clone() const`: returns a deep copy of this node.
- A value-based equality operator. Note that this ignores equality of any annotations.
- `void visit(Visitor &visitor)` (C++ only): implements the visitor pattern using the also-generated abstract Visitor/RecursiveVisitor classes. See below.
- `void dump(std::ostream &out=std::cout, int indent=0)`: does a debug dump of the node to the given stream with the given indentation level.
- `SomeNodeType *as_some_node_type()` (C++ only): does the equivalent of a `dynamic_cast` to the given node type, returning `this` if the type is correct or `nullptr` if not.

Note that the prototypes for the Python equivalents differ as appropriate.

An implicit node class simply named `Node` is always generated, serving as the base class for all other nodes. In C++, it is what derives from the `Base` class defined in the namespace specified using `tree_namespace`. In Python, the `Node` class always derives directly from `object`.

1.4.1 Tree traversal

Tree traversal is accomplished by starting at the root and working your way down. The attributes specified in the tree file appear as public members of the node classes. Maybe, One, OptLink, and Link can be dereferenced simply using the * or -> operators; Any and Many use the [] indexation operator. Note that these are all null- and range-checked, unlike the stdlib equivalents.

Tree traversal is complicated by the fact that you often don’t know exactly what the type of a node is. For example, an expression node may actually be a subtraction or addition. For this, tree-gen supports a few different patterns in C++:

- *Visitor pattern.* You define a class inheriting from the generated `Visitor` or `RecursiveVisitor` classes. These abstract classes provide or require you to implement functions for each node type. The appropriate one then gets called when you pass an instance of your visitor class to the `visit()` method on a node. You must always override `visit_node()`, usually to throw a suitable exception in case an unexpected node is encountered. The default implementation for all the other node types differs between the two classes. For `Visitor`, the default implementation falls back to the next more generically typed function (`visit_subtraction()` falls back to `visit_expression()`, `visit_expression()` falls back to `visit_node()`). For `RecursiveVisitor`, the default implementation for non-leaf node types recursively calls `visit()` for all child nodes, thus recursively traversing the tree. For leaf nodes, both visitor classes have the same behavior.
- *Using the ‘as_()‘ methods.** Given for instance an expression node, you might do

```
if (auto addition = expression.as_addition()) {
  ...
} else if (auto subtraction = expression.as_subtraction()) {
  ...
} else {
  ...
}
```

Because a `nullptr` evaluates to false in C++, the blocks will only be executed if the cast succeeds. Be careful copypasting though; you can accidentally use the `addition` variable in the `subtraction` block if you'd want, but it's obviously null in that case. C++ scoping is weird.

- *Using a switch statement.* You might do

```
switch (expression.type()) {
  case NodeType::Addition:
    ...
  case NodeType::Subtraction:
    ...
}
```

This doesn't handle the cast for you, but in cases where you only need to switch based on the type and don't need access to members of the nodes this is more descriptive than the if/else form.

Just choose the method that makes the most sense within context. Python is so much more dynamic in general that you're better off using its duck typing functionality or using `isinstance()` directly, so no special features are generated for this.

Note that tree-gen trees do *not* contain allow traversal back toward the root of a tree. Supporting this would greatly complicate the internals and the user-facing API, because then you wouldn't be able to just move nodes around. It is therefore key that you design your trees and interfaces such that this information is not needed. If this is somehow impossible, you'll have to manage the links back up the tree manually using (Opt)Link edges.

1.4.2 Serialization and deserialization

Optionally, logic to serialize and deserialize trees can be generated in addition to the APIs above. To enable this, you have to tell tree-gen how to serialize and deserialize the primitive types that you use in the tree through two templated functions that must be specialized for all primitive types (similar and in addition to the initialization function):

```
template <typename T>
void serialize(const T &obj, tree::cbor::MapWriter &map) { ... }

template <typename T>
T deserialize(const tree::cbor::MapReader &map) { ... }
```

The former must serialize `obj` by calling the various `append_*()` functions on the given `tree::cbor::MapWriter` object. The latter must perform the reverse operation. The namespace(s) and names of these functions must be provided to tree-gen using the optional `serdes_functions` directive. Specifying this directive enables the serialization and deserialization logic.

Once enabled, the entry point for serializing a tree is tree::base::serialize() or tree::base::serialize_file(), and deserializing is tree::base::deserialize() or tree::base::deserialize_file(). The internal serialization and deserialization functions in the edge classes and generated node classes are public and could also be used, but these require more boilerplate due to link handling.

If you attempt to use these methods for a tree without serialization/deserialization support enabled, you'll receive a template error. Template errors are notoriously difficult to understand, so you have been warned. Don't use them unless you enable support through the `serdes_functions` directive. Attempting to serialize a tree that is not well-formed will lead to a tree::base::NotWellFormed exception.

Serialization format

The serialization format makes use of the RFC7049 CBOR data representation, serving as a compromise between the readability of JSON and speed/simplicity of the serialization and deserialization logic. Specifically, CBOR can be losslessly converted to and from JSON using third-party tools if need be for debugging (at least for as far as it's used here), but is itself a simple binary format.

Each edge and each primitive receives its own map in the tree. Note that Any and Many are internally represented as a vector of One edges; therefore, Any/Many results in two nested edge objects. The data for nodes and their annotations is stored along with the One/Maybe edges. The keys for the maps are the following:

```
empty Maybe:

    {
        "@T": "?",
        "@t": null
    }

filled Maybe:

    {
        "@T": "?",
        "@i": <sequence number>,
        "@t": "<TitleCase node type name>",
        "<snake_case_attribute_name>": { <attribute data> },
        ...
        "{<annotation type>}": { <annotation data> },
        ...
    }

One:

    {
        "@T": "1",
        "@i": <sequence number>,
        "@t": "<TitleCase node type name>",
        "<snake_case_attribute_name>": { <attribute data> },
        ...
        "{<annotation type>}": { <annotation data> },
        ...
    }

Any:

    {
        "@T": "*",
        "@d": [
            <`One` object for each item>
        ]
    }

Many:

    {
```

(continues on next page)

(continued from previous page)

```
        "@T": "+",
        "@d": [
            <`One` object for each item>
        ]
    }

empty OptLink:

    {
        "@T": "@",
        "@l": null
    }

filled OptLink:

    {
        "@T": "@",
        "@l": <sequence number of linked node>
    }

Link:

    {
        "@T": "$",
        "@l": <sequence number of linked node>
    }
```

The `@T` entry stores which type of edge an object is. This is not strictly needed when deserializing as this information should be known by context; instead, it is used as a validity check. The `@i` sequence numbers are unique integers for all One/Maybe edges in the tree, which are used to recover the (Opt)Link pointers through their `@l` key. The `@t` key is used to recover subtype information, as the tree only knows the abstract supertype of a node in general. These are, however, not necessarily globally unique; they only need to distinguish between subclasses, and therefore just the TitleCase name of the node type without C++ namespace is sufficient.

Keys that start with a { and close with a } are used for annotations. The string enclosed within the { } in the key is used to store the annotation type. The identifier used can either be generated automatically by C++ using `typeid(T).name()`, or can be specified manually to have more control. Any annotation type that doesn't have a serialization and deserialization function registered for it within tree::annotatable::serdes_registry is silently ignored in either direction.

All remaining keys map to the node attributes using their snake_case name. The corresponding value is (recursively) one of the structures above for edges, or a map containing user-specified key/value pairs for primitives, serialized and deserialized using the functions specified by the `serdes_functions` directive.

The names of the keys are chosen such that, when ordered by ASCII value, the order is as specified, and such that there can never be name conflicts. The commonly used keys and values are short to minimize serialization and deserialization overhead.

1.4.3 Python support

In addition to C++, tree-gen can also generate pure-Python objects to represent the tree. As in C++, each node type becomes a class, and class hierarchy is used similarly. The edge classes don't exist, however; instead, their functionality is baked into the generated code to prevent an unnecessary user-facing level of indirection during tree traversal that cannot be hidden in Python. Despite this, the generated structures provide a similar level of type safety as the C++ structures do; all writes to fields of node classes are type-checked.

Where applicable, the generated Python file assumes that the Python module tree corresponds with the namespace hierarchy in the C++ world. This applies to the following things:

- primitive type declarations;
- external node types;
- (de)serialization functions.

The initialization function isn't used in Python; rather, the default constructor of the primitive types is used.

The connection between the C++ and Python world is handled through CBOR serialization and deserialization rather than providing Python wrappers around the C++ objects. While this approach isn't as performant as the alternative, it allows a much simpler and more Pythonic interface to be exposed to the user. Serdes is therefore perhaps the most important feature of the Python generator. Note however that it is entirely up to the program or library using tree-gen to provide the requisite interface between the C++ and Python worlds (for example through swig); tree-gen just ensures that the Python and C++ tree implementations always remain in sync.

The serialization and deserialization functions naturally have a different signature in Python than they do in C++. More specifically, the functions must look like this:

```python
def serialize(typ, val):
    # typ is either a type (for primitives) or a str (for annotations). In
    # the former case, isinstance(val, typ) may be assumed. Return value must
    # be an object consisting of only the following types:
    #  - `dict`s with `str` keys;
    #  - `list`s;
    #  - `int`s between -2^63 and 2^63-1 inclusive;
    #  - `float`s;
    #  - `True`, `False`, or `None`;
    #  - `str`s;
    #  - `bytes` objects.
    # The toplevel object must be a dict.
    pass # TODO

def deserialize(typ, val):
    # typ is either a type (for primitives) or a str (for annotations). val
    # is a structure representing the CBOR serialization using the same
    # Python primitives that can be returned by serialize(). The return value
    # must be an instance of the specified `typ`e for primitives, but can be
    # anything for annotations; for unknown annotation types, just returning
    # val as-is is the recommended fallback behavior.
    pass # TODO
```

Primitive serialization and deserialization can also be done using class methods:

```python
class SomePrimitive:
    def serialize_cbor(self):
        # Called instead of serialize(SomePrimitive, self) if it exists.
        pass # TODO

    @staticmethod
    def deserialize_cbor(cbor):
        # Called instead of deserialize(SomePrimitive, cbor) if it exists.
        pass # TODO
```

If no `serdes_function` directive is specified, serdes logic is still generated on the Python end; in this case, only the class method variant is used, and annotations must be directly serializable to CBOR for them to be serialized (they will be silently ignored if they're not).

CHAPTER 2

Directory tree example

This example illustrates the tree system with a Windows-like directory tree structure. The `System` root node consists of one or more drives, each of which has a drive letter and a root directory with named files and subdirectories. To make things a little more interesting, a symlink-like "mount" is added as a file type, which links to another directory.

Using this tree, this example should teach you the basics of using tree-gen trees. The tutorial runs you through the C++ code of `main.cpp` and Python code of `main.py`, but be sure to check out `directory.tree`, `primitives.hpp`, `primitives.hpp`, and (if you want to reproduce it in your own project) `CMakeLists.txt` copied to the bottom of this page as well. You will also find the complete `main.cpp` and `main.py` there.

Let's start by making the root node using `tree::base::make()`. This works somewhat like `std::make_shared()`, but instead of returning a `shared_ptr` it returns a `One` edge. This might come off as a bit odd, considering trees in graph theory start with a node rather than an edge. This is just a choice, though; a side effect of how the internal ``shared_ptr``s work and how Link/OptLink edges work (if you'd store the tree as the root structure directly, the root node wouldn't always be stored in the same place on the heap, breaking link nodes).

```
auto system = tree::base::make<directory::System>();
```

At all times, you can use the `dump()` method on a node to see what it looks like for debugging purposes. It looks like this:

```
system->dump();
```

↓

```
System(
  drives: !MISSING
)
```

While this system node exists as a tree in memory and tree-gen seems happy, our system tree is at this time not "well-formed". A tree node (or an edge to one) is only considered well-formed if all of the following are true:

- All `One` edges in the tree connect to exactly one node.
- All `Many` edges in the tree connect to at least one node.
- All `Link` and non-empty `OptLink` nodes refer to a node reachable from the root node.

- Each node in the tree is only used by a non-link node once.

Currently, the second requirement is not met.

```
ASSERT(!system.is_well_formed());
```

You can get slightly more information using `check_well_formed()`; instead or returning a boolean, it will throw a `NotWellFormed` exception with an error message.

```
ASSERT_RAISES(tree::base::NotWellFormed, system.check_well_formed());
```

↓

```
tree::base::NotWellFormed exception message: 'Many' edge of type N9directory5DriveE␣
↪is empty
```

Note that the name of the type is “mangled”. The exact output will vary from compiler to compiler, or even from project to project. But hopefully it’ll be readable enough to make sense of in general.

To fix that, let’s add a default drive node to the tree. This should get drive letter `A`, because primitives::initialize() is specialized to return that for **‘‘**Letter‘‘s (see primitives.hpp).

```
system->drives.add(tree::base::make<directory::Drive>());
```

`Drive` has a `One` edge that is no empty, though, so the tree still isn’t well-formed. Let’s add one of those as well.

```
system->drives[0]->root_dir = tree::base::make<directory::Directory>();
```

Now we have a well-formed tree. Let’s have a look:

```
system.check_well_formed();
system->dump();
```

↓

```
System(
  drives: [
    Drive(
      letter: A
      root_dir: <
        Directory(
          entries: []
          name:
        )
      >
    )
  ]
)
```

We can just change the drive letter by assignment, as you would expect.

```
system->drives[0]->letter = 'C';
```

Before we add files and directories, let’s make a shorthand variable for the root directory. Because root_dir is an edge to another node rather than the node itself, and thus acts like a pointer or reference to it, we can just copy it into a variable and modify the variable to update the tree. Note that you’d normally just write “auto” for the type for brevity; the type is shown here to make explicit what it turns into.

```
directory::One<directory::Directory> root = system->drives[0]->root_dir;
```

Let's make a "Program Files" subdirectory and add it to the root.

```
auto programs = tree::base::make<directory::Directory>(
    tree::base::Any<directory::Entry>{},
    "Program Files");
root->entries.add(programs);
system.check_well_formed();
```

That's quite verbose. But in most cases it can be written way shorter. Here's the same with the less versatile but also less verbose emplace() method (which avoids the tree::make() call, but doesn't allow an insertion index to be specified) and with a "using namespace" for the tree. emplace() can also be chained, allowing multiple files and directories to be added at once in this case.

```
{
    using namespace directory;

    root->entries.emplace<Directory>(Any<Entry>{}, "Windows")
                 .emplace<Directory>(Any<Entry>{}, "Users")
                 .emplace<File>("lots of hibernation data", "hiberfil.sys")
                 .emplace<File>("lots of page file data", "pagefile.sys")
                 .emplace<File>("lots of swap data", "swapfile.sys");
}
system.check_well_formed();
```

In order to look for a file in a directory, you'll have to make your own function to iterate over them. After all, tree-gen doesn't know that the name field is a key; it has no concept of a key-value store. This is simple enough to make, but to prevent this example from getting out of hand we'll just use indices for now.

Let's try to read the "lots of data" string from pagefile.sys.

```
ASSERT(root->entries[4]->name == "pagefile.sys");
// ASSERT(root->entries[4]->contents == "lots of page file data");
//   '-> No member named 'contents' in 'directory::Entry'
```

Hm, that didn't work so well, because C++ doesn't know that entry 4 happens to be a file rather than a directory or a mount. We have to tell it to cast to a file first (which throws a std::bad_cast if it's not actually a file). The easiest way to do that is like this:

```
ASSERT(root->entries[4]->as_file()->contents == "lots of page file data");
```

No verbose casts required; tree-gen will make member functions for all the possible subtypes.

While it's possible to put the same node in a tree twice (without using a link), this is not allowed. This isn't checked until a well-formedness check is performed, however (and in fact can't be without having access to the root node).

```
root->entries.add(root->entries[0]);
ASSERT_RAISES(tree::base::NotWellFormed, system.check_well_formed());
```

↓

```
tree::base::NotWellFormed exception message: Duplicate node of type␣
↪N9directory5EntryEat address 0x5588663ab440 found in tree
```

Note that we can index nodes Python-style with negative integers for add() and remove(), so remove(-1) will remove the broken node we just added. Note that the -1 is not actually necessary, though, as it is the default.

```
root->entries.remove(-1);
system.check_well_formed();
```

We *can*, of course, add copies of nodes. That's what copy (shallow) and clone (deep) are for. Usually you'll want a deep copy, but in this case shallow is fine, because a File has no child nodes. Note that, even for a deep copy, links are not modified; it is intended to copy a subtree to another part of the same tree. To make a complete copy of a tree that maintains link semantics (i.e. does not link back to the original tree) the best way would be to go through serialize/deserialize.

```
root->entries.add(root->entries[0]->copy());
system.check_well_formed();
```

Note that the generated classes don't care that there are now two directories named Program Files in the root. As far as they're concerned, they're two different directories with the same name. Let's remove it again, though.

```
root->entries.remove();
```

Something we haven't looked at yet are links. Links are edges in the tree that, well, turn it into something that isn't strictly a tree anymore. While One/Maybe/Any/Many require that nodes are unique, Link/OptLink require that they are *not* unique, and are present elsewhere in the tree. Let's make a new directory, and mount it in the Users directory.

```
auto user_dir = tree::base::make<directory::Directory>(
    tree::base::Any<directory::Entry>{},
    "");
root->entries.emplace<directory::Mount>(user_dir, "SomeUser");
```

Note that user_dir is not yet part of the tree. emplace works simply because it doesn't check whether the directory is in the tree yet. But the tree is no longer well-formed now.

```
ASSERT_RAISES(tree::base::NotWellFormed, system.check_well_formed());
```

↓

```
tree::base::NotWellFormed exception message: Link to node of type␣
↪N9directory9DirectoryE at address 0x5588663abcc0 not found in tree
```

To make it work again, we can add it as a root directory to a second drive.

```
system->drives.emplace<directory::Drive>('D', user_dir);
system.check_well_formed();
```

A good way to confuse a filesystem is to make loops. tree-gen is okay with this, though.

```
system->drives[1]->root_dir->entries.emplace<directory::Mount>(root, "evil link to C:
↪");
system.check_well_formed();
```

The only place where it matters is in the dump function, which only goes one level deep. After that, it'll just print an ellipsis.

```
system->dump();
```

↓

```
System(
  drives: [
```

(continues on next page)

(continued from previous page)

```
    Drive(
      letter: C
      root_dir: <
        Directory(
          entries: [
            Directory(
              entries: []
              name: Program Files
            )
            Directory(
              entries: []
              name: Windows
            )
            Directory(
              entries: []
              name: Users
            )
            File(
              contents: lots of hibernation data
              name: hiberfil.sys
            )
            File(
              contents: lots of page file data
              name: pagefile.sys
            )
            File(
              contents: lots of swap data
              name: swapfile.sys
            )
            Mount(
              target --> <
                Directory(
                  entries: [
                    Mount(
                      target --> <
                        ...
                      >
                      name: evil link to C:
                    )
                  ]
                  name:
                )
              >
              name: SomeUser
            )
          ]
          name:
        )
      >
    )
    Drive(
      letter: D
      root_dir: <
        Directory(
          entries: [
            Mount(
              target --> <
```

(continues on next page)

(continued from previous page)

```
                Directory(
                  entries: [
                    Directory(
                      entries: []
                      name: Program Files
                    )
                    Directory(
                      entries: []
                      name: Windows
                    )
                    Directory(
                      entries: []
                      name: Users
                    )
                    File(
                      contents: lots of hibernation data
                      name: hiberfil.sys
                    )
                    File(
                      contents: lots of page file data
                      name: pagefile.sys
                    )
                    File(
                      contents: lots of swap data
                      name: swapfile.sys
                    )
                    Mount(
                      target --> <
                        ...
                      >
                      name: SomeUser
                    )
                  ]
                  name:
                )
              >
              name: evil link to C:
            )
          ]
          name:
        )
      >
    )
  ]
)
```

Now that we have a nice tree, let's try the serialization and deserialization functionality. Serializing is easy:

```
std::string cbor = tree::base::serialize(system);
```

Let's write it to a file; we'll load this in Python later.

```
{
    std::ofstream cbor_output;
    cbor_output.open("tree.cbor", std::ios::out | std::ios::trunc | std::ios::binary);
    cbor_output << cbor;
}
```

Let's also have a look at a hexdump of that.

```cpp
int count = 0;
for (char c : cbor) {
    if (count == 16) {
        std::printf("\n");
        count = 0;
    } else if (count > 0 && count % 4 == 0) {
        std::printf(" ");
    }
    std::printf("%02X ", (uint8_t)c);
    count++;
}
std::printf("\n");
```

↓

```
BF 62 40 54  61 3F 62 40  69 00 62 40  74 66 53 79
73 74 65 6D  66 64 72 69  76 65 73 BF  62 40 54 61
2B 62 40 64  9F BF 62 40  54 61 31 62  40 69 01 62
40 74 65 44  72 69 76 65  66 6C 65 74  74 65 72 BF
63 76 61 6C  18 43 FF 68  72 6F 6F 74  5F 64 69 72
BF 62 40 54  61 31 62 40  69 02 62 40  74 69 44 69
72 65 63 74  6F 72 79 67  65 6E 74 72  69 65 73 BF
62 40 54 61  2A 62 40 64  9F BF 62 40  54 61 31 62
40 69 03 62  40 74 69 44  69 72 65 63  74 6F 72 79
67 65 6E 74  72 69 65 73  BF 62 40 54  61 2A 62 40
64 9F FF FF  64 6E 61 6D  65 BF 63 76  61 6C 6D 50
72 6F 67 72  61 6D 20 46  69 6C 65 73  FF FF BF 62
40 54 61 31  62 40 69 04  62 40 74 69  44 69 72 65
63 74 6F 72  79 67 65 6E  74 72 69 65  73 BF 62 40
54 61 2A 62  40 64 9F FF  FF 64 6E 61  6D 65 BF 63
76 61 6C 67  57 69 6E 64  6F 77 73 FF  FF BF 62 40
54 61 31 62  40 69 05 62  40 74 69 44  69 72 65 63
74 6F 72 79  67 65 6E 74  72 69 65 73  BF 62 40 54
61 2A 62 40  64 9F FF FF  64 6E 61 6D  65 BF 63 76
61 6C 65 55  73 65 72 73  FF FF BF 62  40 54 61 31
62 40 69 06  62 40 74 64  46 69 6C 65  68 63 6F 6E
74 65 6E 74  73 BF 63 76  61 6C 78 18  6C 6F 74 73
20 6F 66 20  68 69 62 65  72 6E 61 74  69 6F 6E 20
64 61 74 61  FF 64 6E 61  6D 65 BF 63  76 61 6C 6C
68 69 62 65  72 66 69 6C  2E 73 79 73  FF FF BF 62
40 54 61 31  62 40 69 07  62 40 74 64  46 69 6C 65
68 63 6F 6E  74 65 6E 74  73 BF 63 76  61 6C 76 6C
6F 74 73 20  6F 66 20 70  61 67 65 20  66 69 6C 65
20 64 61 74  61 FF 64 6E  61 6D 65 BF  63 76 61 6C
6C 70 61 67  65 66 69 6C  65 2E 73 79  73 FF FF BF
62 40 54 61  31 62 40 69  08 62 40 74  64 46 69 6C
65 68 63 6F  6E 74 65 6E  74 73 BF 63  76 61 6C 71
6C 6F 74 73  20 6F 66 20  73 77 61 70  20 64 61 74
61 FF 64 6E  61 6D 65 BF  63 76 61 6C  6C 73 77 61
70 66 69 6C  65 2E 73 79  73 FF FF BF  62 40 54 61
31 62 40 69  09 62 40 74  65 4D 6F 75  6E 74 66 74
61 72 67 65  74 BF 62 40  54 61 24 62  40 6C 0B FF
64 6E 61 6D  65 BF 63 76  61 6C 68 53  6F 6D 65 55
73 65 72 FF  FF FF FF 64  6E 61 6D 65  BF 63 76 61
6C 60 FF FF  FF BF 62 40  54 61 31 62  40 69 0A 62
40 74 65 44  72 69 76 65  66 6C 65 74  74 65 72 BF
```

(continues on next page)

(continued from previous page)

```
63 76 61 6C  18 44 FF 68  72 6F 6F 74  5F 64 69 72
BF 62 40 54  61 31 62 40  69 0B 62 40  74 69 44 69
72 65 63 74  6F 72 79 67  65 6E 74 72  69 65 73 BF
62 40 54 61  2A 62 40 64  9F BF 62 40  54 61 31 62
40 69 0C 62  40 74 65 4D  6F 75 6E 74  66 74 61 72
67 65 74 BF  62 40 54 61  24 62 40 6C  02 FF 64 6E
61 6D 65 BF  63 76 61 6C  6F 65 76 69  6C 20 6C 69
6E 6B 20 74  6F 20 43 3A  FF FF FF FF  64 6E 61 6D
65 BF 63 76  61 6C 60 FF  FF FF FF FF  FF
```

You can pull that through http://cbor.me and https://jsonformatter.org to inspect the output, if you like.

We can deserialize it into a complete copy of the tree.

```
auto system2 = tree::base::deserialize<directory::System>(cbor);
system2->dump();
```

↓

```
System(
  drives: [
    Drive(
      letter: C
      root_dir: <
        Directory(
          entries: [
            Directory(
              entries: []
              name: Program Files
            )
            Directory(
              entries: []
              name: Windows
            )
            Directory(
              entries: []
              name: Users
            )
            File(
              contents: lots of hibernation data
              name: hiberfil.sys
            )
            File(
              contents: lots of page file data
              name: pagefile.sys
            )
            File(
              contents: lots of swap data
              name: swapfile.sys
            )
            Mount(
              target --> <
                Directory(
                  entries: [
                    Mount(
                      target --> <
                        ...
```

(continues on next page)

(continued from previous page)

```
                    >
                    name: evil link to C:
                  )
                ]
                name:
              )
            >
            name: SomeUser
          )
        ]
        name:
      )
    >
  )
  Drive(
    letter: D
    root_dir: <
      Directory(
        entries: [
          Mount(
            target --> <
              Directory(
                entries: [
                  Directory(
                    entries: []
                    name: Program Files
                  )
                  Directory(
                    entries: []
                    name: Windows
                  )
                  Directory(
                    entries: []
                    name: Users
                  )
                  File(
                    contents: lots of hibernation data
                    name: hiberfil.sys
                  )
                  File(
                    contents: lots of page file data
                    name: pagefile.sys
                  )
                  File(
                    contents: lots of swap data
                    name: swapfile.sys
                  )
                  Mount(
                    target --> <
                      ...
                    >
                    name: SomeUser
                  )
                ]
                name:
              )
            >
```

(continues on next page)

(continued from previous page)

```
              name: evil link to C:
            )
          ]
          name:
        )
      >
    )
  ]
)
```

Note that equality for two link edges is satisfied only if they point to the exact same node. That's not the case for the links in our two entirely separate trees, so the two trees register as unequal.

```
ASSERT(!system2.equals(system));
```

To be sure no data was lost, we'll have to check the CBOR and debug dumps instead.

```
ASSERT(tree::base::serialize(system) == tree::base::serialize(system2));
std::ostringstream ss1{};
system->dump(ss1);
std::ostringstream ss2{};
system2->dump(ss2);
ASSERT(ss1.str() == ss2.str());
```

2.1 Python usage

Let us now turn our attention to tree-gen's Python support. As you may remember from the introduction, tree-gen does not provide a direct interface between Python and C++ using a tool like SWIG; rather, it provides conversion between a CBOR tree representation and the in-memory representations of both languages. It then becomes trivial for users of the tree-gen structure to provide an API to users that converts from one to the other, since only a single string of bytes has to be transferred.

Recall that we wrote the CBOR representation of the tree we constructed in the C++ example to a file. Let's try loading this file with Python.

```
from directory import *

with open(os.path.join(TEST_DIR, 'tree.cbor'), 'rb') as f:
    tree = System.deserialize(f.read())

tree.check_well_formed()
print(tree)
```

↓

```
System(
  drives: [
    Drive(
      letter: C
      root_dir: <
        Directory(
          entries: [
            Directory(
              entries: -
```

(continues on next page)

(continued from previous page)

```
              name: Program Files
            )
            Directory(
              entries: -
              name: Windows
            )
            Directory(
              entries: -
              name: Users
            )
            File(
              contents: lots of hibernation data
              name: hiberfil.sys
            )
            File(
              contents: lots of page file data
              name: pagefile.sys
            )
            File(
              contents: lots of swap data
              name: swapfile.sys
            )
            Mount(
              target --> <
                Directory(
                  entries: [
                    Mount(
                      target --> <
                        ...
                      >
                      name: evil link to C:
                    )
                  ]
                  name:
                )
              >
              name: SomeUser
            )
          ]
          name:
        )
      >
    )
    Drive(
      letter: D
      root_dir: <
        Directory(
          entries: [
            Mount(
              target --> <
                Directory(
                  entries: [
                    Directory(
                      entries: -
                      name: Program Files
                    )
                    Directory(
```

(continues on next page)

(continued from previous page)

```
           entries: -
           name: Windows
          )
          Directory(
           entries: -
           name: Users
          )
          File(
           contents: lots of hibernation data
           name: hiberfil.sys
          )
          File(
           contents: lots of page file data
           name: pagefile.sys
          )
          File(
           contents: lots of swap data
           name: swapfile.sys
          )
          Mount(
           target --> <
            ...
           >
           name: SomeUser
          )
         ]
         name:
        )
       >
       name: evil link to C:
      )
     ]
     name:
    )
   >
  )
 ]
)
```

And there we go; the same structure in pure Python.

As you might expect from a pure-Python structure, field access is much less verbose than it is in C++.

```
print(tree.drives[0].letter)
print(tree.drives[0].root_dir.entries[0].name)
```

↓

```
C
Program Files
```

Perhaps contrary to Python intuition however, the structure provides type-safety. You can't assign values to fields that don't exist…

```
tree.color_of_a_banana = 'yellow'
```

↓

```
Traceback (most recent call last):
  File "main.py", line 48, in <module>
    tree.color_of_a_banana = 'yellow'
AttributeError: 'System' object has no attribute 'color_of_a_banana'
```

. . . or assign values of incorrect types to ones that do:

```
tree.drives[0] = 'not-a-drive'
```

↓

```
Traceback (most recent call last):
  File "main.py", line 55, in <module>
    tree.drives[0] = 'not-a-drive'
  File "/home/docs/checkouts/readthedocs.org/user_builds/tree-gen/checkouts/latest/
↪cbuild/examples/directory/directory.py", line 490, in __setitem__
    .format(val, idx, self._T))
TypeError: object 'not-a-drive' is not an instance of 0
```

Note that the setters will however try to "typecast" objects they don't know about (to be more specific, anything that isn't an instance of Node). So if we try to assign a string directly to tree.drives, you get something you might not have expected:

```
tree.drives = 'not-a-set-of-drives'
```

↓

```
Traceback (most recent call last):
  File "main.py", line 65, in <module>
    tree.drives = 'not-a-set-of-drives'
  File "/home/docs/checkouts/readthedocs.org/user_builds/tree-gen/checkouts/latest/
↪cbuild/examples/directory/directory.py", line 1576, in drives
    val = MultiDrive(val)
  File "/home/docs/checkouts/readthedocs.org/user_builds/tree-gen/checkouts/latest/
↪cbuild/examples/directory/directory.py", line 472, in __init__
    .format(val, idx, self._T))
TypeError: object 'n' at index 0 is not an instance of <class 'directory.Drive'>
```

The setter for the drives property is trying to cast the string to a MultiDrive, which interprets its string input as any other iterable, and ultimately tries to cast the first letter to a Drive. There's not much that can be done about this; it's just how duck typing works.

Manipulation of trees in Python works just as you might expect it to in Python:

```
tree.drives.append(Drive('E'))
tree.drives[-1].root_dir = Directory([
    File('test', 'test-contents'),
    File('test2', 'test2-contents'),
])
del tree.drives[1]
assert not tree.is_well_formed()
tree.drives[0].root_dir.entries[-1].target = tree.drives[1].root_dir
tree.check_well_formed()
print(tree)
```

↓

```
System(
  drives: [
    Drive(
      letter: C
      root_dir: <
        Directory(
          entries: [
            Directory(
              entries: -
              name: Program Files
            )
            Directory(
              entries: -
              name: Windows
            )
            Directory(
              entries: -
              name: Users
            )
            File(
              contents: lots of hibernation data
              name: hiberfil.sys
            )
            File(
              contents: lots of page file data
              name: pagefile.sys
            )
            File(
              contents: lots of swap data
              name: swapfile.sys
            )
            Mount(
              target --> <
                Directory(
                  entries: [
                    File(
                      contents: test
                      name: test-contents
                    )
                    File(
                      contents: test2
                      name: test2-contents
                    )
                  ]
                  name:
                )
              >
              name: SomeUser
            )
          ]
          name:
        )
      >
    )
    Drive(
      letter: E
      root_dir: <
```

(continues on next page)

(continued from previous page)

```
        Directory(
          entries: [
            File(
              contents: test
              name: test-contents
            )
            File(
              contents: test2
              name: test2-contents
            )
          ]
          name:
        )
      >
    )
  ]
)
```

Once the Python code is done manipulating the structure, it can be serialized again.

```python
cbor = tree.serialize()
assert cbor == System.deserialize(cbor).serialize()
count = 0
for c in cbor:
    if count == 16:
        print()
        count = 0
    elif count > 0 and count % 4 == 0:
        print(' ', end='')
    print('%02X ' % c, end='')
    count += 1
print()
```

↓

```
A3 62 40 69  00 62 40 74  66 53 79 73  74 65 6D 66
64 72 69 76  65 73 A2 62  40 54 61 2B  62 40 64 82
A5 62 40 54  61 31 62 40  69 01 62 40  74 65 44 72
69 76 65 66  6C 65 74 74  65 72 A1 63  76 61 6C 18
43 68 72 6F  6F 74 5F 64  69 72 A5 62  40 54 61 31
62 40 69 02  62 40 74 69  44 69 72 65  63 74 6F 72
79 67 65 6E  74 72 69 65  73 A2 62 40  54 61 2A 62
40 64 87 A5  62 40 54 61  31 62 40 69  03 62 40 74
69 44 69 72  65 63 74 6F  72 79 67 65  6E 74 72 69
65 73 A2 62  40 54 61 2A  62 40 64 80  64 6E 61 6D
65 A1 63 76  61 6C 6D 50  72 6F 67 72  61 6D 20 46
69 6C 65 73  A5 62 40 54  61 31 62 40  69 04 62 40
74 69 44 69  72 65 63 74  6F 72 79 67  65 6E 74 72
69 65 73 A2  62 40 54 61  2A 62 40 64  80 64 6E 61
6D 65 A1 63  76 61 6C 67  57 69 6E 64  6F 77 73 A5
62 40 54 61  31 62 40 69  05 62 40 74  69 44 69 72
65 63 74 6F  72 79 67 65  6E 74 72 69  65 73 A2 62
40 54 61 2A  62 40 64 80  64 6E 61 6D  65 A1 63 76
61 6C 65 55  73 65 72 73  A5 62 40 54  61 31 62 40
69 06 62 40  74 64 46 69  6C 65 68 63  6F 6E 74 65
6E 74 73 A1  63 76 61 6C  78 18 6C 6F  74 73 20 6F
66 20 68 69  62 65 72 6E  61 74 69 6F  6E 20 64 61
```

(continues on next page)

(continued from previous page)

```
74 61 64 6E  61 6D 65 A1  63 76 61 6C  6C 68 69 62
65 72 66 69  6C 2E 73 79  73 A5 62 40  54 61 31 62
40 69 07 62  40 74 64 46  69 6C 65 68  63 6F 6E 74
65 6E 74 73  A1 63 76 61  6C 76 6C 6F  74 73 20 6F
66 20 70 61  67 65 20 66  69 6C 65 20  64 61 74 61
64 6E 61 6D  65 A1 63 76  61 6C 6C 70  61 67 65 66
69 6C 65 2E  73 79 73 A5  62 40 54 61  31 62 40 69
08 62 40 74  64 46 69 6C  65 68 63 6F  6E 74 65 6E
74 73 A1 63  76 61 6C 71  6C 6F 74 73  20 6F 66 20
73 77 61 70  20 64 61 74  61 64 6E 61  6D 65 A1 63
76 61 6C 6C  73 77 61 70  66 69 6C 65  2E 73 79 73
A5 62 40 54  61 31 62 40  69 09 62 40  74 65 4D 6F
75 6E 74 64  6E 61 6D 65  A1 63 76 61  6C 68 53 6F
6D 65 55 73  65 72 66 74  61 72 67 65  74 A2 62 40
54 61 24 62  40 6C 0B 64  6E 61 6D 65  A1 63 76 61
6C 60 A5 62  40 54 61 31  62 40 69 0A  62 40 74 65
44 72 69 76  65 66 6C 65  74 74 65 72  A1 63 76 61
6C 18 45 68  72 6F 6F 74  5F 64 69 72  A5 62 40 54
61 31 62 40  69 0B 62 40  74 69 44 69  72 65 63 74
6F 72 79 67  65 6E 74 72  69 65 73 A2  62 40 54 61
2A 62 40 64  82 A5 62 40  54 61 31 62  40 69 0C 62
40 74 64 46  69 6C 65 68  63 6F 6E 74  65 6E 74 73
A1 63 76 61  6C 64 74 65  73 74 64 6E  61 6D 65 A1
63 76 61 6C  6D 74 65 73  74 2D 63 6F  6E 74 65 6E
74 73 A5 62  40 54 61 31  62 40 69 0D  62 40 74 64
46 69 6C 65  68 63 6F 6E  74 65 6E 74  73 A1 63 76
61 6C 65 74  65 73 74 32  64 6E 61 6D  65 A1 63 76
61 6C 6E 74  65 73 74 32  2D 63 6F 6E  74 65 6E 74
73 64 6E 61  6D 65 A1 63  76 61 6C 60
```

2.2 Complete file listings

2.2.1 directory.tree

```
// Attach \file docstrings to the generated files for Doxygen.
# Implementation for classes representing a Windows directory tree.
source

# Header for classes representing a Windows directory tree.
header

// Include tree base classes.
include "tree-base.hpp"
tree_namespace tree::base

// Include primitive types.
include "primitives.hpp"
import primitives

// Initialization function to use to construct default values for the tree base
// classes and primitives.
initialize_function primitives::initialize
serdes_functions primitives::serialize primitives::deserialize
```

(continues on next page)

(continued from previous page)

```

// Set the namespace for the generated classes and attach a docstring.
# Namespace for classes representing a Windows directory tree.
namespace directory

# Root node, containing the drives and associated directory trees.
system {

    # The drives available on the system. There must be at least one.
    drives: Many<drive>;

}

# Represents a drive.
drive {

    # The drive letter used to identify it.
    letter: primitives::Letter;

    # Root directory.
    root_dir: One<directory>;

}

# Represents a directory entry.
entry {

    # Name of the directory entry.
    name: primitives::String;

    # Represents a regular file.
    file {

        # The file contents.
        contents: primitives::String;

    }

    # Represents a (sub)directory.
    directory {

        # The directory contents. Note that directories can be empty.
        entries: Any<entry>;

    }

    # Represents a link to another directory.
    mount {

        # The directory linked to.
        target: Link<directory>;

    }

}
```

2.2.2 primitives.hpp

```cpp
/** \file
 * Defines primitives used in the generated directory tree structure.
 */

#pragma once

#include <string>

/**
 * Namespace with primitives used in the generated directory tree structure.
 */
namespace primitives {

/**
 * Letter primitive, used to represent drive letters.
 */
using Letter = char;

/**
 * Strings, used to represent filenames and file contents.
 */
using String = std::string;

/**
 * Initialization function. This must be specialized for any types used as
 * primitives in a tree that are actual C primitives (int, char, bool, etc),
 * as these are not initialized by the T() construct.
 */
template <class T>
T initialize() { return T(); };

/**
 * Declare the default initializer for drive letters. It's declared inline
 * to avoid having to make a cpp file just for this.
 */
template <>
inline Letter initialize<Letter>() {
    return 'A';
}

/**
 * Serialization function. This must be specialized for any types used as
 * primitives in a tree. The default implementation doesn't do anything.
 */
template <typename T>
void serialize(const T &obj, tree::cbor::MapWriter &map) {
}

/**
 * Serialization function for Letter.
 */
template <>
inline void serialize<Letter>(const Letter &obj, tree::cbor::MapWriter &map) {
    map.append_int("val", obj);
}
```

(continues on next page)

(continued from previous page)

```

/**
 * Serialization function for String.
 */
template <>
inline void serialize<String>(const String &obj, tree::cbor::MapWriter &map) {
    map.append_string("val", obj);
}

/**
 * Deserialization function. This must be specialized for any types used as
 * primitives in a tree. The default implementation doesn't do anything.
 */
template <typename T>
T deserialize(const tree::cbor::MapReader &map) {
    return initialize<T>();
}

/**
 * Deserialization function for Letter.
 */
template <>
inline Letter deserialize<Letter>(const tree::cbor::MapReader &map) {
    return map.at("val").as_int();
}

/**
 * Deserialization function for String.
 */
template <>
inline String deserialize<String>(const tree::cbor::MapReader &map) {
    return map.at("val").as_string();
}

} // namespace primitives
```

2.2.3 primitives.py

```python
class Letter(str):
    """Letter primitive, used to represent drive letters."""
    def __new__(cls, s='A'):
        return str.__new__(cls, s)


class String(str):
    """Strings, used to represent filenames and file contents."""
    pass


def serialize(typ, val):
    """Serialization function."""

    # Serialization formats of primitives.
    if typ is Letter:
        return {'val': ord(val)}
```

(continues on next page)

(continued from previous page)

```
    if typ is String:
        return {'val': val}

    # Serialization formats of annotations.
    if isinstance(typ, str):
        return val

    # Some unknown type.
    raise TypeError('no known serialization of type %r, value %r' % (typ, val))


def deserialize(typ, val):
    """Deserialization function."""

    # Serialization formats of primitives.
    if typ is Letter:
        return Letter(chr(val['val']))
    if typ is String:
        return String(val['val'])

    # Serialization formats of annotations.
    if isinstance(typ, str):
        return val

    # Some unknown type.
    raise TypeError('no known deserialization for type %r, value %r' % (typ, val))
```

2.2.4 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12 FATAL_ERROR)

project(directory-example CXX)

# Add the tree-gen repository root directory. Normally, your project would be
# located outside of tree-gen's root, which means you can omit the second
# argument.
add_subdirectory(../.. tree-gen)

# Generates the files for the directory tree.
generate_tree_py(
    "${CMAKE_CURRENT_SOURCE_DIR}/directory.tree"
    "${CMAKE_CURRENT_BINARY_DIR}/directory.hpp"
    "${CMAKE_CURRENT_BINARY_DIR}/directory.cpp"
    "${CMAKE_CURRENT_BINARY_DIR}/directory.py"
)

add_executable(
    directory-example
    "${CMAKE_CURRENT_BINARY_DIR}/directory.cpp"
    "${CMAKE_CURRENT_SOURCE_DIR}/main.cpp"
)

target_include_directories(
    directory-example
    # This directory for primitives.hpp:
```

(continues on next page)

(continued from previous page)

```
    PRIVATE "${CMAKE_CURRENT_SOURCE_DIR}"
    # Binary directory for directory.hpp:
    PRIVATE "${CMAKE_CURRENT_BINARY_DIR}"
)

target_link_libraries(directory-example tree-lib)

# The following lines only serve to register the example as a test, so you can
# run it using `make test` or `ctest` as well. You don't need them as such in
# your own project.
enable_testing()
add_test(
    NAME directory-example
    COMMAND directory-example
    WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
)

# Only add the Python test if CMake is new enough for us to not have to bother
# with FindPythonInterp.
if(NOT ${CMAKE_VERSION} VERSION_LESS "3.12")
    find_package(Python3 COMPONENTS Interpreter)
    if(${Python3_FOUND})
        add_test(
            NAME directory-example-py
            COMMAND ${Python3_EXECUTABLE} main.py ${CMAKE_CURRENT_BINARY_DIR}
            WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
        )
    endif()
endif()
```

2.2.5 main.cpp

```
#include <iostream>
#include <cstdio>
#include <stdexcept>
#include "../utils.hpp"

// Include the generated file.
#include "directory.hpp"

// Note: the // comment contents of main(), together with the MARKER lines and
// the output of the program, are used to automatically turn this into a
// restructured-text page for ReadTheDocs.

int main() {

    // **********************
    // Directory tree example
    // **********************
    //
    // This example illustrates the tree system with a Windows-like directory
    // tree structure. The ``System`` root node consists of one or more drives,
    // each of which has a drive letter and a root directory with named files
    // and subdirectories. To make things a little more interesting, a
    // symlink-like "mount" is added as a file type, which links to another
```

(continues on next page)

(continued from previous page)

```
    // directory.
    //
    // Using this tree, this example should teach you the basics of using
    // tree-gen trees. The tutorial runs you through the C++ code of
    // ``main.cpp`` and Python code of ``main.py``, but be sure to check out
    // ``directory.tree``, ``primitives.hpp``, ``primitives.hpp``, and (if you
    // want to reproduce it in your own project) ``CMakeLists.txt`` copied to
    // the bottom of this page as well. You will also find the complete
    // ``main.cpp`` and ``main.py`` there.
    //
    // Let's start by making the root node using ``tree::base::make()``. This
    // works somewhat like ``std::make_shared()``, but instead of returning a
    // ``shared_ptr`` it returns a ``One`` edge. This might come off as a bit
    // odd, considering trees in graph theory start with a node rather than an
    // edge. This is just a choice, though; a side effect of how the internal
    // ``shared_ptr``s work and how Link/OptLink edges work (if you'd store the
    // tree as the root structure directly, the root node wouldn't always be
    // stored in the same place on the heap, breaking link nodes).
    auto system = tree::base::make<directory::System>();
    MARKER

    // At all times, you can use the ``dump()`` method on a node to see what it
    // looks like for debugging purposes. It looks like this:
    system->dump();
    MARKER

    // While this system node exists as a tree in memory and tree-gen seems
    // happy, our system tree is at this time not "well-formed". A tree node (or
    // an edge to one) is only considered well-formed if all of the following
    // are true:
    //
    //  - All ``One`` edges in the tree connect to exactly one node.
    //  - All ``Many`` edges in the tree connect to at least one node.
    //  - All ``Link`` and non-empty ``OptLink`` nodes refer to a node reachable
    //    from the root node.
    //  - Each node in the tree is only used by a non-link node once.
    //
    // Currently, the second requirement is not met.
    ASSERT(!system.is_well_formed());
    MARKER

    // You can get slightly more information using ``check_well_formed()``;
    // instead or returning a boolean, it will throw a ``NotWellFormed``
    // exception with an error message.
    ASSERT_RAISES(tree::base::NotWellFormed, system.check_well_formed());
    MARKER

    // Note that the name of the type is
    // `"mangled" <https://en.wikipedia.org/wiki/Name_mangling#C++>`_. The exact
    // output will vary from compiler to compiler, or even from project to
    // project. But hopefully it'll be readable enough to make sense of in
    // general.

    // To fix that, let's add a default drive node to the tree. This should get
    // drive letter ``A``, because primitives::initialize() is specialized to
    // return that for ``Letter``s (see primitives.hpp).
    system->drives.add(tree::base::make<directory::Drive>());
```

(continues on next page)

(continued from previous page)

```
    MARKER

    // ``Drive`` has a ``One`` edge that is no empty, though, so the tree still
    // isn't well-formed. Let's add one of those as well.
    system->drives[0]->root_dir = tree::base::make<directory::Directory>();
    MARKER

    // Now we have a well-formed tree. Let's have a look:
    system.check_well_formed();
    system->dump();
    MARKER

    // We can just change the drive letter by assignment, as you would expect.
    system->drives[0]->letter = 'C';
    MARKER

    // Before we add files and directories, let's make a shorthand variable for
    // the root directory. Because root_dir is an edge to another node rather
    // than the node itself, and thus acts like a pointer or reference to it,
    // we can just copy it into a variable and modify the variable to update the
    // tree. Note that you'd normally just write "auto" for the type for
    // brevity; the type is shown here to make explicit what it turns into.
    directory::One<directory::Directory> root = system->drives[0]->root_dir;
    MARKER

    // Let's make a "Program Files" subdirectory and add it to the root.
    auto programs = tree::base::make<directory::Directory>(
        tree::base::Any<directory::Entry>{},
        "Program Files");
    root->entries.add(programs);
    system.check_well_formed();
    MARKER

    // That's quite verbose. But in most cases it can be written way shorter.
    // Here's the same with the less versatile but also less verbose emplace()
    // method (which avoids the tree::make() call, but doesn't allow an
    // insertion index to be specified) and with a "using namespace" for the
    // tree. emplace() can also be chained, allowing multiple files and
    // directories to be added at once in this case.
    {
        using namespace directory;

        root->entries.emplace<Directory>(Any<Entry>{}, "Windows")
                     .emplace<Directory>(Any<Entry>{}, "Users")
                     .emplace<File>("lots of hibernation data", "hiberfil.sys")
                     .emplace<File>("lots of page file data", "pagefile.sys")
                     .emplace<File>("lots of swap data", "swapfile.sys");
    }
    system.check_well_formed();
    MARKER

    // In order to look for a file in a directory, you'll have to make your own
    // function to iterate over them. After all, tree-gen doesn't know that the
    // name field is a key; it has no concept of a key-value store. This is
    // simple enough to make, but to prevent this example from getting out of
    // hand we'll just use indices for now.

```

(continues on next page)

(continued from previous page)

```
    // Let's try to read the "lots of data" string from pagefile.sys.
    ASSERT(root->entries[4]->name == "pagefile.sys");
    // ASSERT(root->entries[4]->contents == "lots of page file data");
    //  '-> No member named 'contents' in 'directory::Entry'
    MARKER

    // Hm, that didn't work so well, because C++ doesn't know that entry 4
    // happens to be a file rather than a directory or a mount. We have to tell
    // it to cast to a file first (which throws a std::bad_cast if it's not
    // actually a file). The easiest way to do that is like this:
    ASSERT(root->entries[4]->as_file()->contents == "lots of page file data");
    MARKER

    // No verbose casts required; tree-gen will make member functions for all
    // the possible subtypes.

    // While it's possible to put the same node in a tree twice (without using
    // a link), this is not allowed. This isn't checked until a well-formedness
    // check is performed, however (and in fact can't be without having access
    // to the root node).
    root->entries.add(root->entries[0]);
    ASSERT_RAISES(tree::base::NotWellFormed, system.check_well_formed());
    MARKER

    // Note that we can index nodes Python-style with negative integers for
    // add() and remove(), so remove(-1) will remove the broken node we just
    // added. Note that the -1 is not actually necessary, though, as it is the
    // default.
    root->entries.remove(-1);
    system.check_well_formed();
    MARKER

    // We *can*, of course, add copies of nodes. That's what copy (shallow) and
    // clone (deep) are for. Usually you'll want a deep copy, but in this case
    // shallow is fine, because a File has no child nodes. Note that, even for
    // a deep copy, links are not modified; it is intended to copy a subtree to
    // another part of the same tree. To make a complete copy of a tree that
    // maintains link semantics (i.e. does not link back to the original tree)
    // the best way would be to go through serialize/deserialize.
    root->entries.add(root->entries[0]->copy());
    system.check_well_formed();
    MARKER

    // Note that the generated classes don't care that there are now two
    // directories named Program Files in the root. As far as they're concerned,
    // they're two different directories with the same name. Let's remove it
    // again, though.
    root->entries.remove();
    MARKER

    // Something we haven't looked at yet are links. Links are edges in the tree
    // that, well, turn it into something that isn't strictly a tree anymore.
    // While One/Maybe/Any/Many require that nodes are unique, Link/OptLink
    // require that they are *not* unique, and are present elsewhere in the
    // tree. Let's make a new directory, and mount it in the Users directory.
    auto user_dir = tree::base::make<directory::Directory>(
        tree::base::Any<directory::Entry>{},
```

(continues on next page)

(continued from previous page)

```
        "");
    root->entries.emplace<directory::Mount>(user_dir, "SomeUser");
    MARKER

    // Note that user_dir is not yet part of the tree. emplace works simply
    // because it doesn't check whether the directory is in the tree yet. But
    // the tree is no longer well-formed now.
    ASSERT_RAISES(tree::base::NotWellFormed, system.check_well_formed());
    MARKER

    // To make it work again, we can add it as a root directory to a second
    // drive.
    system->drives.emplace<directory::Drive>('D', user_dir);
    system.check_well_formed();
    MARKER

    // A good way to confuse a filesystem is to make loops. tree-gen is okay
    // with this, though.
    system->drives[1]->root_dir->entries.emplace<directory::Mount>(root, "evil link␣
↪to C:");
    system.check_well_formed();
    MARKER

    // The only place where it matters is in the dump function, which only goes
    // one level deep. After that, it'll just print an ellipsis.
    system->dump();
    MARKER

    // Now that we have a nice tree, let's try the serialization and
    // deserialization functionality. Serializing is easy:
    std::string cbor = tree::base::serialize(system);
    MARKER

    // Let's write it to a file; we'll load this in Python later.
    {
        std::ofstream cbor_output;
        cbor_output.open("tree.cbor", std::ios::out | std::ios::trunc |␣
↪std::ios::binary);
        cbor_output << cbor;
    }
    MARKER

    // Let's also have a look at a hexdump of that.
    int count = 0;
    for (char c : cbor) {
        if (count == 16) {
            std::printf("\n");
            count = 0;
        } else if (count > 0 && count % 4 == 0) {
            std::printf(" ");
        }
        std::printf("%02X ", (uint8_t)c);
        count++;
    }
    std::printf("\n");
    MARKER

```

(continues on next page)

(continued from previous page)

```
    // You can pull that through http://cbor.me and https://jsonformatter.org
    // to inspect the output, if you like.

    // We can deserialize it into a complete copy of the tree.
    auto system2 = tree::base::deserialize<directory::System>(cbor);
    system2->dump();
    MARKER

    // Note that equality for two link edges is satisfied only if they point to
    // the exact same node. That's not the case for the links in our two
    // entirely separate trees, so the two trees register as unequal.
    ASSERT(!system2.equals(system));
    MARKER

    // To be sure no data was lost, we'll have to check the CBOR and debug dumps
    // instead.
    ASSERT(tree::base::serialize(system) == tree::base::serialize(system2));
    std::ostringstream ss1{};
    system->dump(ss1);
    std::ostringstream ss2{};
    system2->dump(ss2);
    ASSERT(ss1.str() == ss2.str());
    MARKER

    return 0;
}
```

2.2.6 main.py

```
import sys, os, traceback
TEST_DIR = os.path.realpath(sys.argv[1])
sys.path.append(TEST_DIR)


def marker():
    print('###MARKER###')


# Note: the # | comment contents of this file, together with the marker() lines
# and the output of the program, are used to automatically turn this into a
# restructured-text page for ReadTheDocs. The output is appended to the C++
# output.

# | Python usage
# | ============

# | Let us now turn our attention to tree-gen's Python support. As you may
# | remember from the introduction, tree-gen does not provide a direct interface
# | between Python and C++ using a tool like SWIG; rather, it provides
# | conversion between a CBOR tree representation and the in-memory
# | representations of both languages. It then becomes trivial for users of the
# | tree-gen structure to provide an API to users that converts from one to the
# | other, since only a single string of bytes has to be transferred.

# | Recall that we wrote the CBOR representation of the tree we constructed in
```

(continues on next page)

(continued from previous page)

```
# | the C++ example to a file. Let's try loading this file with Python.
from directory import *

with open(os.path.join(TEST_DIR, 'tree.cbor'), 'rb') as f:
    tree = System.deserialize(f.read())

tree.check_well_formed()
print(tree)
marker()

# | And there we go; the same structure in pure Python.

# | As you might expect from a pure-Python structure, field access is much less
# | verbose than it is in C++.
print(tree.drives[0].letter)
print(tree.drives[0].root_dir.entries[0].name)
marker()

# | Perhaps contrary to Python intuition however, the structure provides
# | type-safety. You can't assign values to fields that don't exist...
try:
    tree.color_of_a_banana = 'yellow'
except AttributeError:
    traceback.print_exc(file=sys.stdout)
marker()

# | ... or assign values of incorrect types to ones that do:
try:
    tree.drives[0] = 'not-a-drive'
except TypeError:
    traceback.print_exc(file=sys.stdout)
marker()

# | Note that the setters will however try to "typecast" objects they don't know
# | about (to be more specific, anything that isn't an instance of Node). So if
# | we try to assign a string directly to tree.drives, you get something you
# | might not have expected:
try:
    tree.drives = 'not-a-set-of-drives'
except TypeError:
    traceback.print_exc(file=sys.stdout)
marker()

# | The setter for the drives property is trying to cast the string to a
# | MultiDrive, which interprets its string input as any other iterable, and
# | ultimately tries to cast the first letter to a Drive. There's not much that
# | can be done about this; it's just how duck typing works.

# | Manipulation of trees in Python works just as you might expect it to in
# | Python:
tree.drives.append(Drive('E'))
tree.drives[-1].root_dir = Directory([
    File('test', 'test-contents'),
    File('test2', 'test2-contents'),
])
del tree.drives[1]
assert not tree.is_well_formed()
```

(continues on next page)

(continued from previous page)

```
tree.drives[0].root_dir.entries[-1].target = tree.drives[1].root_dir
tree.check_well_formed()
print(tree)
marker()

# | Once the Python code is done manipulating the structure, it can be
# | serialized again.
cbor = tree.serialize()
assert cbor == System.deserialize(cbor).serialize()
count = 0
for c in cbor:
    if count == 16:
        print()
        count = 0
    elif count > 0 and count % 4 == 0:
        print(' ', end='')
    print('%02X ' % c, end='')
    count += 1
print()
marker()
```

CHAPTER 3

Interpreter example

This example discusses some of the more advanced features of tree-gen, using an interpreter for a very simple language as an example. This is, however, not a completely integrated or even functional interpreter; we just go over some of the concepts and things you may run into when you would make such an interpreter.

3.1 External nodes & multiple trees

When trees get large and complicated, you may want to split a tree up into multiple files. This is supported within tree-gen, as long as there are no circular references to nodes between the files. In this example, we've split up the expression-like nodes into a tree separate from the statement-like nodes. The former, `value.tree`, has nothing special going on in this regard, while the latter, `program.tree`, effectively includes the former. The files are listed at the bottom of this page for your convenience.

tree-gen itself is unaware of any links between files. For `program.tree`, it simply generates code assuming that the types referred to through the `external` edges exist. For this to be true, you have to tell tree-gen to include the external tree's generated header file at the top of its generated header file using the `include` directive. Note also that, as tree-gen is unaware of the external tree, you'll have to use the full C++ TitleCase type name, rather than the snake_case name for normal edges.

3.2 Integration with Flex/Bison (or other parsers)

When making an interpreter (or parser of any kind) with tree-gen, you'll probably want to use a tokenizer/parser generator such as Flex/Bison. With Bison at least, you'll quickly run into a fundamental issue when trying to store tree-gen nodes in the generated `yyval` union: union in C++ can only consist of types with trivial constructors, and tree-gen nodes are not among those. The author is unfamiliar with other frameworks like it, but this will probably apply for any C-style parser generator.

The solution is to use raw pointers to the generated node types in this enumeration. For instance, you may end up with the following (which would be generated by Bison):

```
union node_type {
    program::Program *prog;
    program::Assignment *asgn;
    value::Literal *lit;
    value::Reference *ref;
    // ...
};
```

To create a new node, instead of using `tree::base::make`, you use the `new` keyword directly.

```
node_type n1, n2;
n1.lit = new value::Literal(2);
n2.ref = new value::Reference("x");
```

The generated constructors won't help you much for non-leaf nodes, because they expect edge wrappers (One, Maybe, etc.) around the parameters, so you'll typically want to use the constructor without arguments. Instead, to assign child nodes, you can use `Maybe::set_raw()`, `One::set_raw()`, `Any::add_raw()`, or `Many::add_raw()`. These functions take ownership of a new-allocated raw pointer to a node - exactly what we need here. Note that because they take ownership, you don't have to (and shouldn't!) delete manually.

```
node_type n3;
n3.asgn = new program::Assignment();
n3.asgn->rhs.set_raw(n1.lit);
n3.asgn->lhs.set_raw(n2.ref);

node_type n4;
n4.prog = new program::Program();
n4.prog->statements.add_raw(n3.asgn);
```

Bison will ultimately pass you the final parsed node, which you then have to put into a `One` edge yourself:

```
auto tree = tree::base::One<program::Program>();
tree.set_raw(n4.prog);
tree->dump();
```

↓

```
Program(
  variables: []
  statements: [
    Assignment(
      lhs: <
        Reference(
          name: x
          target --> !MISSING
        )
      >
      rhs: <
        Literal(
          value: 2
        )
      >
    )
  ]
)
```

Note that when you're using Bison, the enum and its types is largely hidden from you. In the actions the above would look more like this:

```
// Action for a literal:
{ $$ = new value::Literal(std::stoi($1)); }

// Action for a reference:
{ $$ = new value::Reference($1); /* and deallocate $1, assuming it's a char* */ }

// Action for a assignment:
{ $$ = new value::Assignment(); $$->lhs.set_raw($1); $$->rhs.set_raw($3); }
```

and so on.

3.2.1 Syntax error recovery

Some parser generators (like Bison) allow you to specify recovery rules in case of a syntax error, so the parser doesn't just die immediately. This is necessary to emit more than a single error message at a time. To help you store recovery information in the tree for post-mortem analysis, tree-gen has a special `error` directive that you can place inside a node class. For example, if you make a recovery rule that assumes any semicolon encountered after a syntax error produces a broken statement, you can make a node like this:

```
# An erroneous statement.
erroneous_statement {
    error;
}
```

Such nodes behave exactly like any other node, with one exception: they always indicate that they're not well-formed.

```
auto err_stmt = tree::base::make<program::ErroneousStatement>();
ASSERT_RAISES(tree::base::NotWellFormed, err_stmt.check_well_formed());
```

↓

```
tree::base::NotWellFormed exception message: ErroneousStatement error node in tree
```

3.2.2 Line number information (and annotations in general)

When you're doing any kind of serious parsing, you'll want to store as much line number and other contextual information as possible, to make your error messages as clear as possible. You could of course add this information to every node in the tree specification file, perhaps using inheritance to prevent excessive repetition, but this gets annoying fast, especially when it comes to tree-gen's well-formedness rules.

Instead, you should be using annotations for this. Annotations are objects added to nodes without you ever having to declare that you're going to add them. If that sounds like black magic to you in the context of C++, well, that's because it is: internally annotations are stored as a map from `std::type_index` to a C++11 backport of `std::any`... it's complicated. But you don't have to worry about it. What matters is that each and every node generated by tree-gen (or anything else that inherits from `tree::annotatable::Annotatable`) can have zero or one annotation of every C++ type attached to it.

The use of annotations is not limited to metadata (although that is its primary purpose); they also allow bits of code to temporarily attach their own data to tree nodes, without the tree definitions needing to be updated to reflect this. This is especially useful when you're using tree-gen in a library, and you want to let users operate on your trees. After all, they wouldn't be able to modify the tree definition file without forking and recompiling your library.

tree-gen has only one special case for annotations, intended for adding line number information to debug dumps. The `location` directive is used for this:

```
// Source location annotation type. The generated dumper will see if this
// annotation exists for a node, and if so, write it to the debug dump using
// the << stream operator.
location primitives::SourceLocation
```

Without going into too much detail about the Annotatable interface (just look at the API docs for that), here's an example of how it would work.

```
n1.lit->set_annotation(primitives::SourceLocation("test", 1, 5));
n2.ref->set_annotation(primitives::SourceLocation("test", 1, 1));
n3.asgn->set_annotation(primitives::SourceLocation("test", 1, 1));
n4.prog->set_annotation(primitives::SourceLocation("test", 1, 1));
tree->dump();
```

↓

```
Program( # test:1:1
  variables: []
  statements: [
    Assignment( # test:1:1
      lhs: <
        Reference( # test:1:1
          name: x
          target --> !MISSING
        )
      >
      rhs: <
        Literal( # test:1:5
          value: 2
        )
      >
    )
  ]
)
```

Note that we can still use these pointers despite ownership having been passed to the node objects because they refer to the same piece of memory. In practice, though, you would do this during the parser actions, just after constructing them.

When serializing and deserializing, annotations are ignored by default; they can be of any type whatsoever, and C++ can't dynamically introspect which types can be (de)serialized and which can't be, after all. So even though the example SourceLocation object extends `tree::annotatable::Serializable`, this doesn't work automagically.

```
auto node = tree::base::make<value::Literal>(2);
node->set_annotation(primitives::SourceLocation("test", 1, 1));
std::string cbor = tree::base::serialize(node);
auto node2 = tree::base::deserialize<value::Literal>(cbor);
ASSERT(!node2->has_annotation<primitives::SourceLocation>());

// However, you *can* register annotations types for serialization and
// deserialization if you want to through the
// ``tree::annotatable::serdes_registry`` singleton. After that, it will
// work.
tree::annotatable::serdes_registry.add<primitives::SourceLocation>("loc");
cbor = tree::base::serialize(node);
node2 = tree::base::deserialize<value::Literal>(cbor);
ASSERT(node2->has_annotation<primitives::SourceLocation>());
```

Two `add()` methods are provided. The one used here assumes that the type has an appropriate `serialize()` method and an appropriate constructor for deserialization, the other allows you to specify them manually using ‘‘std::function‘‘s. The name is optional but strongly recommended; if not used, a C++-compiler-specific identifier will be used for the type.

Similar to links, annotations aren't copied as you might expect by `copy()` and `clone()`. Specifically, annotations are stored as `std::shared_ptr``s to unknown C++ objects, and therefore copying a node only copies the references to the annotations. To fully clone annotations, you'll either have to serialize and deserialize the node they belong to (after registering with ``serdes_registry` of course), or clone them manually.

3.3 Visitor pattern

In the directory example, we avoided the difficulty of dealing with edges to incomplete types, such as a `One<value::Rvalue>`, aside from a single `as_file()` call halfway through. Spamming such typecasts to, for instance, evaluate an rvalue expression during constant propagation or interpreting, is not very scalable. It can also lead to headache down the line if you ever have to add more subclasses, as it's easy to forget or not bother to check for unknown types when initially writing that code that way. tree-gen also generates the necessary classes for the visitor pattern for you to avoid this.

In the visitor pattern, you define a class that implements the appropriate visitor interface, and (in this case) the nodes will call methods on this class depending on their type when you pass the visitor object to them through their `visit()` method. Because the visitor interface classes are also generated by tree-gen and have sane default behavior, your code will "by default" be future-proof, in the sense that you'll get an error if something unexpected happens, rather than it failing silently and maybe crashing down the line.

Two visitor base classes are provided, `Visitor<T=void>` and `RecursiveVisitor`. In either case, you have to override `visit_node()`, which is called when a node is encountered for which no specialization is available; this should just throw an appropriate exception. You can then optionally override any node in the node class hierarchy to provide the actual behavior.

The two classes differ in the default behavior when an unspecialized node is encountered: `Visitor` always defers up the class hierarchy, while `RecursiveVisitor` provides a default implementation for node types with edges, where it simply recursively visits the child nodes depth-first.

The T type in `Visitor<T>` refers to the return type for the visit methods. It's fixed to `void` for `RecursiveVisitor`, because if a node has multiple children, there is nothing sensible to return.

Let's look at what an expression evaluator looks like for this.

```
class RvalueEvaluator : public value::Visitor<int> {
public:
    int visit_node(value::Node &node) override {
        throw std::runtime_error("unknown node type");
    }

    int visit_literal(value::Literal &node) override {
        return node.value;
    }

    int visit_negate(value::Negate &node) override {
        return -node.oper->visit(*this);
    }

    int visit_add(value::Add &node) override {
        return node.lhs->visit(*this) + node.rhs->visit(*this);
```

(continues on next page)

(continued from previous page)

```
    }

    int visit_sub(value::Sub &node) override {
        return node.lhs->visit(*this) - node.rhs->visit(*this);
    }

    int visit_mul(value::Mul &node) override {
        return node.lhs->visit(*this) * node.rhs->visit(*this);
    }

    int visit_div(value::Div &node) override {
        return node.lhs->visit(*this) / node.rhs->visit(*this);
    }

    int visit_reference(value::Reference &node) override {
        return node.target->value;
    }
};

// 2 + (3 * 4) = 14
auto expr = tree::base::make<value::Add>(
    tree::base::make<value::Literal>(2),
    tree::base::make<value::Mul>(
        tree::base::make<value::Literal>(3),
        tree::base::make<value::Literal>(4)
    )
);
RvalueEvaluator eval{};
ASSERT(expr->visit(eval) == 14);
```

If we wouldn't have implemented `visit_node()`, the `RvalueEvaluator eval{};` line would break, as RvalueEvaluator would be an abstract class because of it. And while the above may look pretty complete, it is in fact not:

```
ASSERT_RAISES(std::runtime_error, value::ErroneousValue().visit(eval));
```

↓

```
std::runtime_error exception message: unknown node type
```

The visitor pattern is more powerful than recursively calling functions for other reasons as well, because they can be specialized through inheritance, state can be maintained in the class as the tree is traversed (the `dump()` method is actually implemented this way internally), it can be written inline within a function despite the recursive nature, and so on. But it's a lot more verbose than the `as_*()` alternative for simple cases. The choice between visitor pattern and casts is therefore usually a matter of personal preference.

3.4 Complete file listings

3.4.1 value.tree

```
// Attach \file docstrings to the generated files for Doxygen.
# Implementation for value tree classes.
source

```

(continues on next page)

(continued from previous page)

```
# Header for value tree classes.
header

// Include tree base classes.
include "tree-base.hpp"
tree_namespace tree::base

// Include primitive types.
include "primitives.hpp"
import primitives

// Initialization function to use to construct default values for the tree base
// classes and primitives.
initialize_function primitives::initialize
serdes_functions primitives::serialize primitives::deserialize

// Source location annotation type. The generated dumper will see if this
// annotation exists for a node, and if so, write it to the debug dump using
// the << stream operator.
location primitives::SourceLocation

// Set the namespace for the generated classes and attach a docstring.
# Namespace for value tree classes.
namespace value

# Variable instance.
variable {

    # The name of the variable.
    name: primitives::Str;

    # The current value of the variable while interpreting.
    value: primitives::Int;

}

# Toplevel expression node.
rvalue {

    # A literal integer.
    literal {

        # The value of the literal.
        value: primitives::Int;

    }

    # Unary operators.
    unop {

        # The operand.
        oper: One<rvalue>;

        # Negation.
        negate {}

    }
```

(continues on next page)

(continued from previous page)

```

    # Binary operators.
    binop {

        # Left-hand side of the expression.
        lhs: One<rvalue>;

        # Right-hand side of the expression.
        rhs: One<rvalue>;

        # Addition.
        add {}

        # Subtraction.
        sub {}

        # Multiplication.
        mul {}

        # Division.
        div {}

    }

    # Toplevel node for assignable expressions.
    lvalue {

        # A variable reference.
        reference {

            # The name used to refer to the variable.
            name: primitives::Str;

            # The variable being referenced.
            // Note that we use a link here to allow multiple places in the
            // tree to refer to the same variable instance in our toy
            // interpreter, so we don't have to do a hashmap lookup each time
            // and a well-formed tree by definition cannot refer to variables
            // that don't exist.
            target: Link<variable>;

        }

        # An erroneous expression.
        erroneous_value {
            error;
        }

    }

}
```

3.4.2 program.tree

```
// Attach \file docstrings to the generated files for Doxygen.
# Implementation for program tree classes.
source

# Header for program tree classes.
header

// Include tree base classes.
include "tree-base.hpp"
tree_namespace tree::base

// Include primitive types.
include "primitives.hpp"
import primitives

// Include the generated value tree header.
include "value.hpp"
import value

// Initialization function to use to construct default values for the tree base
// classes and primitives.
initialize_function primitives::initialize
serdes_functions primitives::serialize primitives::deserialize

// Source location annotation type. The generated dumper will see if this
// annotation exists for a node, and if so, write it to the debug dump using
// the << stream operator.
location primitives::SourceLocation

// Set the namespace for the generated classes and attach a docstring.
# Namespace for program tree classes.
namespace program

# A program is a collection of statements.
program {

    # Variable declarations for the program.
    variables: external Any<value::Variable>;

    # The statements in the program.
    statements: Any<statement>;

}


# A statement.
statement {

    # Conditional statement.
    if_else {

        # Branch condition.
        cond: external One<value::Rvalue>;

        # If block.
```

(continues on next page)

(continued from previous page)

```
        if_block: Any<statement>;

        # Else block.
        else_block: Any<statement>;

    }

    # For loop, from start to (non-inclusive) end.
    for_loop {

        # Variable to assign.
        var: external One<value::Lvalue>;

        # Start value.
        start: external One<value::Rvalue>;

        # Stop value.
        stop: external One<value::Rvalue>;

        # The repeated code.
        block: Any<statement>;

    }

    # Assignment statement.
    assignment {

        # Variable to assign.
        lhs: external One<value::Lvalue>;

        # Expression to assign it to.
        rhs: external One<value::Rvalue>;

    }

    # Print statement.
    print {

        # Value to print.
        expr: external One<value::Rvalue>;

    }

    # An erroneous statement.
    erroneous_statement {
        error;
    }

}
```

3.4.3 primitives.hpp

```
/** \file
 * Defines primitives used in the generated directory tree structure.
 */
```

(continues on next page)

(continued from previous page)

```

#pragma once

#include <string>
#include "tree-base.hpp"

/**
 * Namespace with primitives used in the generated directory tree structure.
 */
namespace primitives {

/**
 * Integer primitive.
 */
using Int = int;

/**
 * Strings, used to represent filenames and file contents.
 */
using Str = std::string;

/**
 * Initialization function. This must be specialized for any types used as
 * primitives in a tree that are actual C primitives (int, char, bool, etc),
 * as these are not initialized by the T() construct.
 */
template <class T>
T initialize() { return T(); };

/**
 * Declare the default initializer for integers. It's declared inline to avoid
 * having to make a cpp file just for this.
 */
template <>
inline Int initialize<Int>() {
    return 0;
}

/**
 * Serialization function. This must be specialized for any types used as
 * primitives in a tree. The default implementation doesn't do anything.
 */
template <typename T>
void serialize(const T &obj, tree::cbor::MapWriter &map) {
}

/**
 * Serialization function for Int.
 */
template <>
inline void serialize<Int>(const Int &obj, tree::cbor::MapWriter &map) {
    map.append_int("val", obj);
}

/**
 * Serialization function for Str.
 */
```

(continues on next page)

(continued from previous page)

```
template <>
inline void serialize<Str>(const Str &obj, tree::cbor::MapWriter &map) {
    map.append_string("val", obj);
}

/**
 * Deserialization function. This must be specialized for any types used as
 * primitives in a tree. The default implementation doesn't do anything.
 */
template <typename T>
T deserialize(const tree::cbor::MapReader &map) {
    return initialize<T>();
}

/**
 * Deserialization function for Int.
 */
template <>
inline Int deserialize<Int>(const tree::cbor::MapReader &map) {
    return map.at("val").as_int();
}

/**
 * Deserialization function for Str.
 */
template <>
inline Str deserialize<Str>(const tree::cbor::MapReader &map) {
    return map.at("val").as_string();
}

/**
 * Source location annotation object, containing source file line numbers etc.
 */
class SourceLocation: tree::annotatable::Serializable {
public:
    std::string filename;
    int line;
    int column;

    SourceLocation(
        const std::string &filename,
        uint32_t line = 0,
        uint32_t column = 0
    ) : filename(filename), line(line), column(column) {
    }

    /**
     * Serialization logic for SourceLocation.
     */
    inline void serialize(tree::cbor::MapWriter &map) const override {
        map.append_string("filename", filename);
        map.append_int("line", line);
        map.append_int("column", column);
    }

    /**
     * Deserialization logic for SourceLocation.
```

(continues on next page)

(continued from previous page)

```
     */
    inline SourceLocation(const tree::cbor::MapReader &map) {
        filename = map.at("filename").as_string();
        line = map.at("line").as_int();
        column = map.at("column").as_int();
    }

};

/**
 * Stream << overload for source location objects.
 */
inline std::ostream& operator<<(std::ostream& os, const primitives::SourceLocation&␣
↪object) {
    os << object.filename << ":" << object.line << ":" << object.column;
    return os;
}

} // namespace primitives
```

3.4.4 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12 FATAL_ERROR)

project(interpreter-example CXX)

# Add the tree-gen repository root directory. Normally, your project would be
# located outside of tree-gen's root, which means you can omit the second
# argument.
add_subdirectory(../.. tree-gen)

# Generates the files for the value tree.
generate_tree_py(
    "${CMAKE_CURRENT_SOURCE_DIR}/value.tree"
    "${CMAKE_CURRENT_BINARY_DIR}/value.hpp"
    "${CMAKE_CURRENT_BINARY_DIR}/value.cpp"
    "${CMAKE_CURRENT_BINARY_DIR}/value.py"
)

# Generates the files for the program tree.
generate_tree_py(
    "${CMAKE_CURRENT_SOURCE_DIR}/program.tree"
    "${CMAKE_CURRENT_BINARY_DIR}/program.hpp"
    "${CMAKE_CURRENT_BINARY_DIR}/program.cpp"
    "${CMAKE_CURRENT_BINARY_DIR}/program.py"
)

add_executable(
    interpreter-example
    "${CMAKE_CURRENT_BINARY_DIR}/value.cpp"
    "${CMAKE_CURRENT_BINARY_DIR}/program.cpp"
    "${CMAKE_CURRENT_SOURCE_DIR}/main.cpp"
)

target_include_directories(
```

(continues on next page)

(continued from previous page)

```
    interpreter-example
    # This directory for primitives.hpp:
    PRIVATE "${CMAKE_CURRENT_SOURCE_DIR}"
    # Binary directory for directory.hpp:
    PRIVATE "${CMAKE_CURRENT_BINARY_DIR}"
)

target_link_libraries(interpreter-example tree-lib)

# The following lines only serve to register the example as a test, so you can
# run it using `make test` or `ctest` as well. You don't need them as such in
# your own project.
enable_testing()
add_test(interpreter-example interpreter-example)
```

3.4.5 main.cpp

```
#include <iostream>
#include <cstdio>
#include <stdexcept>
#include "../utils.hpp"

// Include the generated files.
#include "value.hpp"
#include "program.hpp"

// Note: the // comment contents of main(), together with the MARKER lines and
// the output of the program, are used to automatically turn this into a
// restructured-text page for ReadTheDocs.

int main() {

    // ******************
    // Interpreter example
    // ******************
    //
    // This example discusses some of the more advanced features of tree-gen,
    // using an interpreter for a very simple language as an example. This is,
    // however, not a completely integrated or even functional interpreter; we
    // just go over some of the concepts and things you may run into when you
    // would make such an interpreter.
    //
    // External nodes & multiple trees
    // ===============================
    //
    // When trees get large and complicated, you may want to split a tree up
    // into multiple files. This is supported within tree-gen, as long as there
    // are no circular references to nodes between the files. In this example,
    // we've split up the expression-like nodes into a tree separate from the
    // statement-like nodes. The former, ``value.tree``, has nothing special
    // going on in this regard, while the latter, ``program.tree``, effectively
    // includes the former. The files are listed at the bottom of this page
    // for your convenience.
    //
    // tree-gen itself is unaware of any links between files. For
```

(continues on next page)

(continued from previous page)

```
    // ``program.tree``, it simply generates code assuming that the types
    // referred to through the ``external`` edges exist. For this to be true,
    // you have to tell tree-gen to include the external tree's generated
    // header file at the top of its generated header file using the ``include``
    // directive. Note also that, as tree-gen is unaware of the external tree,
    // you'll have to use the full C++ TitleCase type name, rather than the
    // snake_case name for normal edges.
    //
    // Integration with Flex/Bison (or other parsers)
    // ==============================================
    //
    // When making an interpreter (or parser of any kind) with tree-gen, you'll
    // probably want to use a tokenizer/parser generator such as Flex/Bison.
    // With Bison at least, you'll quickly run into a fundamental issue when
    // trying to store tree-gen nodes in the generated ``yyval`` union:
    // union in C++ can only consist of types with trivial constructors, and
    // tree-gen nodes are not among those. The author is unfamiliar with
    // other frameworks like it, but this will probably apply for any C-style
    // parser generator.
    //
    // The solution is to use raw pointers to the generated node types in this
    // enumeration. For instance, you may end up with the following (which would
    // be generated by Bison):
    union node_type {
        program::Program *prog;
        program::Assignment *asgn;
        value::Literal *lit;
        value::Reference *ref;
        // ...
    };
    MARKER

    // To create a new node, instead of using ``tree::base::make``,
    // you use the ``new`` keyword directly.
    node_type n1, n2;
    n1.lit = new value::Literal(2);
    n2.ref = new value::Reference("x");
    MARKER

    // The generated constructors won't help you much for non-leaf nodes,
    // because they expect edge wrappers (One, Maybe, etc.) around the
    // parameters, so you'll typically want to use the constructor without
    // arguments. Instead, to assign child nodes, you can use
    // ``Maybe::set_raw()``, ``One::set_raw()``, ``Any::add_raw()``,
    // or ``Many::add_raw()``. These functions take ownership of a new-allocated
    // raw pointer to a node - exactly what we need here. Note that because they
    // take ownership, you don't have to (and shouldn't!) delete manually.
    node_type n3;
    n3.asgn = new program::Assignment();
    n3.asgn->rhs.set_raw(n1.lit);
    n3.asgn->lhs.set_raw(n2.ref);

    node_type n4;
    n4.prog = new program::Program();
    n4.prog->statements.add_raw(n3.asgn);
    MARKER

```

(continues on next page)

(continued from previous page)

```
    // Bison will ultimately pass you the final parsed node, which you then have
    // to put into a ``One`` edge yourself:
    auto tree = tree::base::One<program::Program>();
    tree.set_raw(n4.prog);
    tree->dump();
    MARKER

    // Note that when you're using Bison, the enum and its types is largely
    // hidden from you. In the actions the above would look more like this:
    //
    // .. code-block:: none
    //
    //   // Action for a literal:
    //   { $$ = new value::Literal(std::stoi($1)); }
    //
    //   // Action for a reference:
    //   { $$ = new value::Reference($1); /* and deallocate $1, assuming it's a char*␣
↪*/ }
    //
    //   // Action for a assignment:
    //   { $$ = new value::Assignment(); $$->lhs.set_raw($1); $$->rhs.set_raw($3); }
    //
    // and so on.
    //
    // Syntax error recovery
    // ---------------------
    //
    // Some parser generators (like Bison) allow you to specify recovery rules
    // in case of a syntax error, so the parser doesn't just die immediately.
    // This is necessary to emit more than a single error message at a time.
    // To help you store recovery information in the tree for post-mortem
    // analysis, tree-gen has a special ``error`` directive that you can place
    // inside a node class. For example, if you make a recovery rule that
    // assumes any semicolon encountered after a syntax error produces a broken
    // statement, you can make a node like this:
    //
    // .. code-block:: none
    //
    //   # An erroneous statement.
    //   erroneous_statement {
    //       error;
    //   }
    //
    // Such nodes behave exactly like any other node, with one exception: they
    // always indicate that they're not well-formed.
    auto err_stmt = tree::base::make<program::ErroneousStatement>();
    ASSERT_RAISES(tree::base::NotWellFormed, err_stmt.check_well_formed());
    MARKER;

    // Line number information (and annotations in general)
    // ----------------------------------------------------
    //
    // When you're doing any kind of serious parsing, you'll want to store as
    // much line number and other contextual information as possible, to make
    // your error messages as clear as possible. You could of course add this
    // information to every node in the tree specification file, perhaps using
    // inheritance to prevent excessive repetition, but this gets annoying fast,
```

(continues on next page)

(continued from previous page)

```
    // especially when it comes to tree-gen's well-formedness rules.
    //
    // Instead, you should be using annotations for this. Annotations are
    // objects added to nodes without you ever having to declare that you're
    // going to add them. If that sounds like black magic to you in the context
    // of C++, well, that's because it is: internally annotations are stored
    // as a map from ``std::type_index`` to a C++11 backport of ``std::any``...
    // it's complicated. But you don't have to worry about it. What matters is
    // that each and every node generated by tree-gen (or anything else that
    // inherits from ``tree::annotatable::Annotatable``) can have zero or one
    // annotation of every C++ type attached to it.
    //
    // The use of annotations is not limited to metadata (although that is its
    // primary purpose); they also allow bits of code to temporarily attach
    // their own data to tree nodes, without the tree definitions needing to be
    // updated to reflect this. This is especially useful when you're using
    // tree-gen in a library, and you want to let users operate on your trees.
    // After all, they wouldn't be able to modify the tree definition file
    // without forking and recompiling your library.
    //
    // tree-gen has only one special case for annotations, intended for adding
    // line number information to debug dumps. The ``location`` directive is
    // used for this:
    //
    // .. code-block:: none
    //
    //   // Source location annotation type. The generated dumper will see if this
    //   // annotation exists for a node, and if so, write it to the debug dump using
    //   // the << stream operator.
    //   location primitives::SourceLocation
    //
    // Without going into too much detail about the Annotatable interface (just
    // look at the API docs for that), here's an example of how it would work.
    n1.lit->set_annotation(primitives::SourceLocation("test", 1, 5));
    n2.ref->set_annotation(primitives::SourceLocation("test", 1, 1));
    n3.asgn->set_annotation(primitives::SourceLocation("test", 1, 1));
    n4.prog->set_annotation(primitives::SourceLocation("test", 1, 1));
    tree->dump();
    MARKER

    // Note that we can still use these pointers despite ownership having been
    // passed to the node objects because they refer to the same piece of
    // memory. In practice, though, you would do this during the parser actions,
    // just after constructing them.
    //
    // When serializing and deserializing, annotations are ignored by default;
    // they can be of any type whatsoever, and C++ can't dynamically introspect
    // which types can be (de)serialized and which can't be, after all. So even
    // though the example SourceLocation object extends
    // ``tree::annotatable::Serializable``, this doesn't work automagically.
    auto node = tree::base::make<value::Literal>(2);
    node->set_annotation(primitives::SourceLocation("test", 1, 1));
    std::string cbor = tree::base::serialize(node);
    auto node2 = tree::base::deserialize<value::Literal>(cbor);
    ASSERT(!node2->has_annotation<primitives::SourceLocation>());

    // However, you *can* register annotations types for serialization and
```

(continues on next page)

(continued from previous page)

```
    // deserialization if you want to through the
    // ``tree::annotatable::serdes_registry`` singleton. After that, it will
    // work.
    tree::annotatable::serdes_registry.add<primitives::SourceLocation>("loc");
    cbor = tree::base::serialize(node);
    node2 = tree::base::deserialize<value::Literal>(cbor);
    ASSERT(node2->has_annotation<primitives::SourceLocation>());
    MARKER

    // Two ``add()`` methods are provided. The one used here assumes that the
    // type has an appropriate ``serialize()`` method and an appropriate
    // constructor for deserialization, the other allows you to specify them
    // manually using ``std::function``s. The name is optional but strongly
    // recommended; if not used, a C++-compiler-specific identifier will be used
    // for the type.
    //
    // Similar to links, annotations aren't copied as you might expect by
    // ``copy()`` and ``clone()``. Specifically, annotations are stored as
    // ``std::shared_ptr``s to unknown C++ objects, and therefore copying a
    // node only copies the references to the annotations. To fully clone
    // annotations, you'll either have to serialize and deserialize the node
    // they belong to (after registering with ``serdes_registry`` of course), or
    // clone them manually.

    // Visitor pattern
    // ===============
    //
    // In the directory example, we avoided the difficulty of dealing with edges
    // to incomplete types, such as a ``One<value::Rvalue>``, aside from a
    // single ``as_file()`` call halfway through. Spamming such typecasts to,
    // for instance, evaluate an rvalue expression during constant propagation
    // or interpreting, is not very scalable. It can also lead to headache down
    // the line if you ever have to add more subclasses, as it's easy to forget
    // or not bother to check for unknown types when initially writing that code
    // that way. tree-gen also generates the necessary classes for the visitor
    // pattern for you to avoid this.
    //
    // In the visitor pattern, you define a class that implements the
    // appropriate visitor interface, and (in this case) the nodes will call
    // methods on this class depending on their type when you pass the visitor
    // object to them through their ``visit()`` method. Because the visitor
    // interface classes are also generated by tree-gen and have sane default
    // behavior, your code will "by default" be future-proof, in the sense that
    // you'll get an error if something unexpected happens, rather than it
    // failing silently and maybe crashing down the line.
    //
    // Two visitor base classes are provided, ``Visitor<T=void>`` and
    // ``RecursiveVisitor``. In either case, you have to override
    // ``visit_node()``, which is called when a node is encountered for which
    // no specialization is available; this should just throw an appropriate
    // exception. You can then optionally override any node in the node class
    // hierarchy to provide the actual behavior.
    //
    // The two classes differ in the default behavior when an unspecialized
    // node is encountered: ``Visitor`` always defers up the class hierarchy,
    // while ``RecursiveVisitor`` provides a default implementation for node
    // types with edges, where it simply recursively visits the child nodes
```

(continues on next page)

(continued from previous page)

```
    // depth-first.
    //
    // The T type in ``Visitor<T>`` refers to the return type for the visit
    // methods. It's fixed to ``void`` for ``RecursiveVisitor``, because if a
    // node has multiple children, there is nothing sensible to return.
    //
    // Let's look at what an expression evaluator looks like for this.
    class RvalueEvaluator : public value::Visitor<int> {
    public:
        int visit_node(value::Node &node) override {
            throw std::runtime_error("unknown node type");
        }

        int visit_literal(value::Literal &node) override {
            return node.value;
        }

        int visit_negate(value::Negate &node) override {
            return -node.oper->visit(*this);
        }

        int visit_add(value::Add &node) override {
            return node.lhs->visit(*this) + node.rhs->visit(*this);
        }

        int visit_sub(value::Sub &node) override {
            return node.lhs->visit(*this) - node.rhs->visit(*this);
        }

        int visit_mul(value::Mul &node) override {
            return node.lhs->visit(*this) * node.rhs->visit(*this);
        }

        int visit_div(value::Div &node) override {
            return node.lhs->visit(*this) / node.rhs->visit(*this);
        }

        int visit_reference(value::Reference &node) override {
            return node.target->value;
        }
    };

    // 2 + (3 * 4) = 14
    auto expr = tree::base::make<value::Add>(
        tree::base::make<value::Literal>(2),
        tree::base::make<value::Mul>(
            tree::base::make<value::Literal>(3),
            tree::base::make<value::Literal>(4)
        )
    );
    RvalueEvaluator eval{};
    ASSERT(expr->visit(eval) == 14);
    MARKER

    // If we wouldn't have implemented ``visit_node()``, the
    // ``RvalueEvaluator eval{};`` line would break, as RvalueEvaluator would
    // be an abstract class because of it. And while the above may look pretty
```

(continues on next page)

(continued from previous page)

```
    // complete, it is in fact not:
    ASSERT_RAISES(std::runtime_error, value::ErroneousValue().visit(eval));
    MARKER

    // The visitor pattern is more powerful than recursively calling functions
    // for other reasons as well, because they can be specialized through
    // inheritance, state can be maintained in the class as the tree is
    // traversed (the ``dump()`` method is actually implemented this way
    // internally), it can be written inline within a function despite the
    // recursive nature, and so on. But it's a lot more verbose than the
    // ``as_*()`` alternative for simple cases. The choice between visitor
    // pattern and casts is therefore usually a matter of personal preference.
    MARKER

    return 0;
}
```