Los Alamos
NATIONAL LABORATORY
EST.1943

LA-UR-20-29611
Approved for public release; distribution is unlimited.

Title: GNDStk (GNDS Toolkit)

Author(s): Staley, Martin Frank

Intended for: Report

Issued: 2020-11-20

Disclaimer:
Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by Triad National Security, LLC for the National Nuclear Security Administration of U.S. Department of Energy under contract 89233218CNA000001. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

GNDStk (GNDS Toolkit)

Martin Staley

Special thanks to: Jeremy Conlin, Wim Haeck, Nathan Gibson

Objectives, Part I

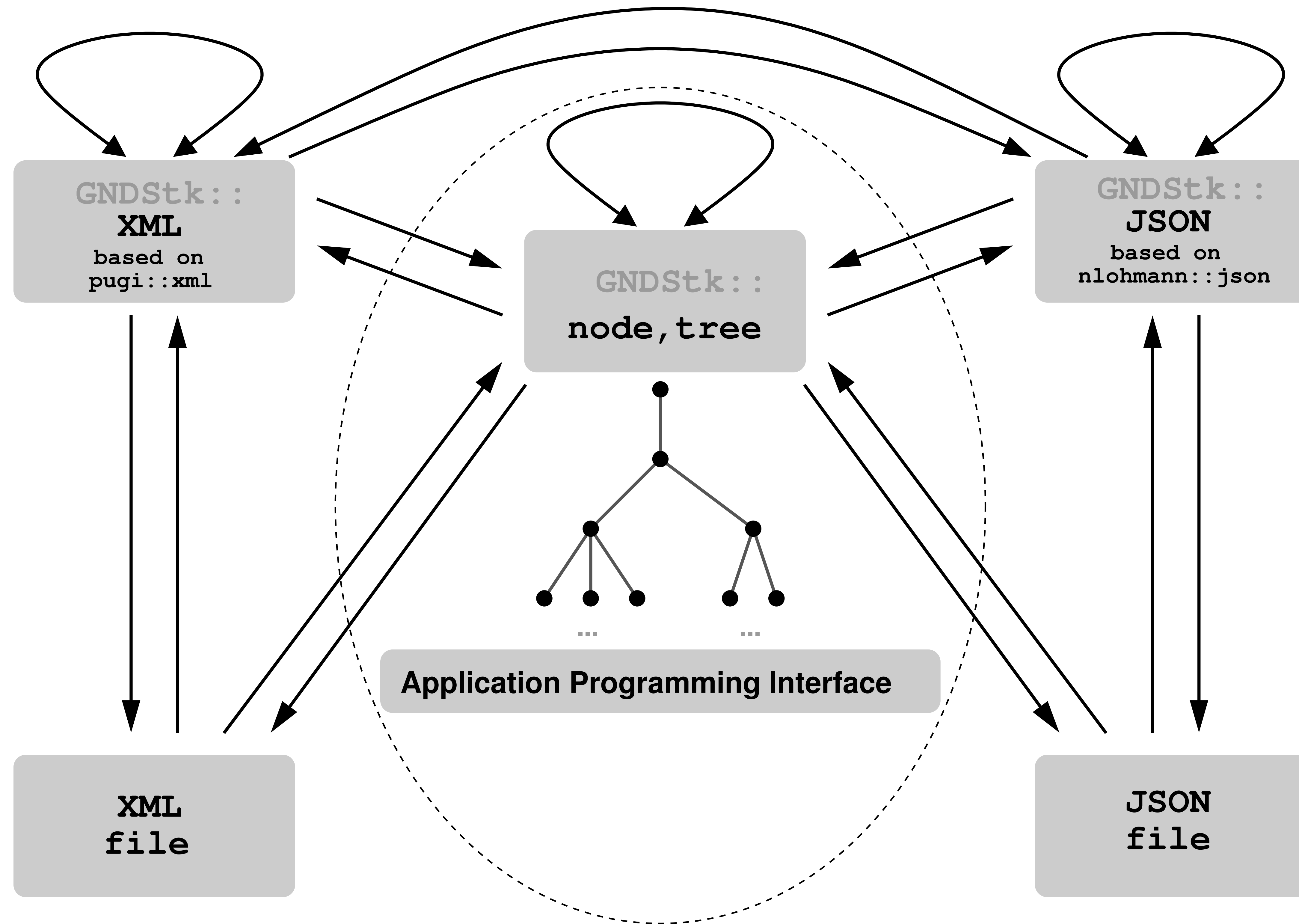
Create a software toolkit for working with GNDS data

- **Generic internal C++ structures** for GNDS data, independent of what external file formats are involved.
- **GNDS versions**. Support the existing GNDS standard, as well as any and all future versions.
- **Read** GNDS files. **XML** and **JSON** formats are supported now. We plan to support **HDF5**, and others as may be needed eventually.
- **Modify** existing GNDS data.
- **Create** new GNDS data.
- **Write** GNDS files: XML, JSON, etc.
- **Convert** between file types.
- **Notation**: clean, concise, uncluttered, intuitive.
- **Queries**: search/navigate down into a GNDS data tree.
- **Creation/Modifications**: similar notation to that for queries, but for adding or modifying data.

Objectives, Part II

- **Extensible (and fun) "smart data access" scheme**.
- **Code interoperability** made easy: interface with nuclear data codes, user codes, other GNDS libraries, viz, etc.
- **Modern C++**. Use new language features when (and – importantly – only when) warranted.
- **Header-only library**: make code compilation simple.
- **Simple build system**: don't foist a build battle onto users.
- **Fast Compilation**. You won't wait minutes or hours ~~to get error messages~~ for your code to compile.
- **Leverage** good existing C++ libraries for XML, JSON, HDF5, etc.
- **Python wrappers** for users who prefer Python over C++.
- **GNDStk-based command-line tools**, similar to what we have for ENDF.
- **Test suite**, **tutorial**, **and examples**.
- **Full documentation**.

High-Level Sketch



Examples 1 & 2: Reading and Writing GNDS

Example 1

```
#include "GNDStk.hpp"
using namespace njoy::GNDStk;

int main()
{
   tree t("n-092_U_235.xml");

   // Alternative:
   // tree t;
   // t.read("n-092_U_235.xml");
}
```

Example 2

```
#include "GNDStk.hpp"
using namespace njoy::GNDStk;

int main()
{
   // Read from XML
   tree t("n-092_U_235.xml");

   // Write to JSON
   t.write("n-092_U_235.json");

   // Read back from JSON
   tree j("n-092_U_235.json");

   // You get the idea!
}
```

Example 3: Raw Data Access

This example illustrates how you can access GNDStk's data structures directly

Some content from `n-092_U_235.xml`:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <reactionSuite projectile="n" target="U235" evaluation=...>
3    <styles>
4      ...
5    </styles>
6    <externalFiles>
7      ...
8    </externalFiles>
9    <documentations>
10     ...
11   </documentations>
12   <PoPs name="protare_internal" version="1.0" format="0.1">
13     ...
14   </PoPs>
15   ...
16 </reactionSuite>
```

Example 3

```
1  #include "GNDStk.hpp"
2  using namespace njoy::GNDStk;
3
4  int main()
5  {
6     tree t("n-092_U_235.xml");
7     node n = t.top(); // Top-level GNDS node
8
9     // Name
10    assert(n.name == "reactionSuite");
11
12    // Metadata: projectile, target, ...
13    assert(n.metadata.size() == 5);
14    assert(n.metadata[0].first  == "projectile");
15    assert(n.metadata[0].second == "n");
16    assert(n.metadata[1].first  == "target");
17    assert(n.metadata[1].second == "U235");
18    assert(n.metadata[2].first  == "evaluation");
19    assert(n.metadata[2].second == "ENDF/B-8.0");
20    assert(n.metadata[3].first  == "format");
21    assert(n.metadata[3].second == "1.9");
22    assert(n.metadata[4].first  == "projectileFrame");
23    assert(n.metadata[4].second == "lab");
24
25    // Child nodes: <styles>, <externalFiles>, ...
26    assert(n.children.size() == 9);
27    assert(n.children[0]->name == "styles");
28    assert(n.children[1]->name == "externalFiles");
29    assert(n.children[2]->name == "documentations");
30    assert(n.children[2]->name == "PoPs");
31    // ...
32 }
```

However, most users should avoid code like the above; we have something much better!

And by the way: generally, avoid [] indexing for accessing data. Can you guess why?

Example 4: Improved Data Access

We've built a very flexible and user extensible data access system

```
1  #include "GNDStk.hpp"
2  using namespace njoy::GNDStk;
3
4  // Get a set of basic "smart-access objects",
5  // for both metadata and child nodes
6  using namespace basic;
7
8  int main()
9  {
10    // Let's make the tree const, just to be different
11    // from before. (And we'll only read, not write.)
12    const tree t("n-092_U_235.xml");
13
14    // We know that this GNDS hierarchy's top-level
15    // node is called reactionSuite, so start there.
16    assert(t(reactionSuite).metadata.size() == 5);
17    assert(t(reactionSuite).children.size() == 9);
18
19    // Access reactionSuite's metadata:
20    assert(t(reactionSuite,projectile) == "n");
21    assert(t(reactionSuite,target) == "U235");
22    assert(t(reactionSuite,evaluation) == "ENDF/B-8.0");
23    assert(t(reactionSuite,format) == "1.9");
24    assert(t(reactionSuite,projectileFrame) == "lab");
25
26    // Now let's dig a bit deeper, into the PoPs node:
27    assert(t(reactionSuite,PoPs,name) == "protare_internal");
28    assert(t(reactionSuite,PoPs,version) == "1.0");
29    assert(t(reactionSuite,PoPs,format) == "0.1");
30 }
```

Some remarks:

Namespace `njoy::GNDStk::basic` contains **`reactionSuite`**, **`PoPs`**, etc. to represent GNDS nodes.

Namespace `njoy::GNDStk::basic` contains **`projectile`**, **`target`**, etc. to represent GNDS metadata.

They're *variables* of sorts - notice that they aren't quoted - and you can make your own.

Let's talk about how all this works...

Sketch: Our Data-Query System

Brief recap of what we've spoken about before

One might imagine a basic query into the GNDS tree that looks something like:

```
tree.child(‘‘reactionSuite‘‘).child(‘‘PoPs‘‘).meta(‘‘version‘‘);
```

A somewhat obvious improvement is to predefine the strings, so we can write:

```
tree.child(reactionSuite).child(PoPs).meta(version);
```

Next, we distinguish child nodes and metadata in a compile-time manner, so that overloads of `operator()` can replace `.child()` and `.meta()`:

```
tree(reactionSuite)(PoPs)(version);
```

Next, we generalize our operator() so it can take multiple arguments in sequence:

```
tree(reactionSuite,PoPs,version); // returns a string; say, ‘‘1.0’’
```

Finally, version could encode a type (say, `double`), so that instead of **`‘‘1.0’’`** we get **`1.0`**. (Actually, **`version`** might deserve a type with `major,minor,patch`.)

Data-Query “Keywords”

Our data-query types are **meta_t** and **child_t**. (Might be renamed.) From these you can make data-query “keywords” - just a term we use for variables of these types.

```
meta_t< TYPE, CONVERTER >
   Var( ‘‘GNDSname’’, converter );
```

Var: Variable you use in queries. Might well be GNDSname, but doesn’t need to be.

GNDSname: Name in GNDS, e.g. “version” for the `version` GNDS metadatum.

TYPE: Return object of this type. Use void to indicate raw tree form – a `std::string`.

CONVERTER: Optional custom conversion from raw form (`std::string`) to TYPE. (Analogy: giving `std::sort()` a custom comparison.)

Define a function `convert(std::string,TYPE&)` for a custom TYPE, or use a converter object that takes `(std::string,TYPE&)`.

```
child_t< TYPE, ALLOW, CONVERTER, FILTER >
   Var( ‘‘GNDSname’’, converter, filter );
```

Var, GNDSname: as with meta_t. Here, TYPE void means `GNDStk::Node` – a raw tree node.

ALLOW: Does GNDS allow child nodes named “GNDSname” to appear allow::**one** time or allow::**many** times under a given parent?

CONVERTER: Optional custom conversion from `GNDStk::Node` to TYPE.

Define `convert(GNDStk::Node,TYPE&)`, or use a converter that takes `(GNDStk::Node,TYPE&)`.

FILTER: Optional requirement that a node must satisfy. Should take `(GNDStk::Node)` and return `true` or `false`.

Discussion

- The quoted name (“GNDSname”) can be a **C++ regex-style regular expression**.
- We have constructs to make it **easier to build meta_t/child_t** objects. Example:

```
auto my_custom_converter = [](auto &str, double &v) { /* compute v */ };
// <CONVERTER> (my_custom_converter's type) can be determined automatically
// for the meta_t that's created here, because a function call is involved...
auto myversion = keyword.meta<double>("version", my_custom_converter);
```

- The `allow::one` vs. `allow::many` distinction is important!
- It roughly aligns with the distinction between `scalars` and `vectors` in mathematics, and determines if a query will return `one object`, or a `container`.
- So: be aware of how the child_t objects you use are encoded.
- The distinction exists in GNDS; we're simply reflecting it.

Discussion (continued)

- **General form** (almost) of our Tree/Node operator():

  (any number of **initial parameters** ... , **final parameter**)

  Initial parameters:

  - An `allow::one` child_t optionally followed by a string.
  - An `allow::many` child_t followed (*not* optionally) by a string. GNDStk tries to match the string with a `label` metadatum.

  Final parameter:

  - `meta_t`. ...Query returns one value.
  - `allow::one child_t`. ...Query returns one value.
  - `allow::many child_t`. ...Query returns a vector of values.

Example 5: Portion of XML, and Longer Query

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <reactionSuite projectile="n" target="U235">
3
4    <styles>
5      ...
6    </styles>
7
8    <reactions>
9
10     <reaction ...>
11     </reaction>
12
13     <reaction ...>
14     </reaction>
15
16     ...
17
18     <reaction label="U236 + photon" ENDF_MT="102">
19
20       <crossSection>
21
22         <resonancesWithBackground label="eval">
23           ...
24         </resonancesWithBackground>
25
26         <XYs1d label="recon">
27           <axes>
28             <axis index="1" label="energy_in" unit="eV"/>
29             <axis index="0" label="crossSection" unit="b"/>
30           </axes>
31
32           <values>
33             1.00000000e-05 5.72271748e+03 1.09045319e-05 ...
34             1.00000000e-05 5.72271748e+03 1.09045319e-05 ...
35             3.27518236e-04 9.99177575e+02 3.59212441e-04 ...
36             1.55234071e-02 1.33891859e+02 1.63837870e-02 ...
37             2.46706690e-01 4.66440392e+01 2.53371770e-01 ...
38             7.36784047e-01 8.08390701e+00 7.53695364e-01 ...
39             1.15449358e+00 3.79655577e+01 1.15956698e+00 ...
40             1.53945474e+00 4.41233118e+00 1.56418700e+00 ...
41             1.97616679e+00 1.41140575e+01 1.97754515e+00 ...
42             ...
43           </values>
44
45         </XYs1d>
46
47       </crossSection>
48       <outputChannel genre="NBody">
49       </outputChannel>
50     </reaction>
51   </reactions>
52 </reactionSuite>
```

Example lookup, illustrating operator():

```
1  #include "GNDStk.hpp"
2  using namespace njoy::GNDStk;
3  using namespace basic; // basic set of meta_ts and child_ts
4
5  int main()
6  {
7     const tree u235("n-092_U_235.xml");
8     auto data = u235(
9        reactionSuite,
10       reactions,
11       reaction, "U236 + photon", // get the so-labeled <reaction>
12       crossSection,
13       XYs1d, "recon", // get the so-labeled <XYs1d>
14       values<double>
15    );
16 }
```

Experimental “Keyword Arithmetic”

This is brand new, experimental, and perhaps somewhat stylized and unusual

```
 meta_t<TYPE,CONVERTER> var("name")
*       T/   /C              /"new"
        -    --
             post
```

```
 child_t<TYPE,ALLOW,CONVERTER,FILTER> var("name")
*        T/   ++    /C        +F          /"new"
         -    --    --
              pre   post
```

`*var`	Make name regex match-anything
`T/var`	Change TYPE to T
`-var`	Change TYPE to void
`var/C`	Change CONVERTER to C
`var--`	Remove custom converter
`var/"new"`	Change "name" to new name "new"

`*var`	Make name regex match-anything
`T/var`	Change TYPE to T
`-var`	Change TYPE to void
`++var`	Upgrade ALLOW to allow::many
`--var`	Downgrade ALLOW to allow::one
`var/C`	Change CONVERTER to C
`var--`	Remove custom converter
`var+F`	Change FILTER to F
`var/"new"`	Change "name" to new name "new"

There’s a logical pattern to these; you just need to get the hang of it!

Example 6

```
1  #include "GNDStk.hpp"
2  using namespace njoy::GNDStk;
3
4  // Basic set of meta_ts and child_ts
5  using namespace basic;
6
7  // Generic <void,allow::one> child_t
8  const child_t<> seed("name won’t matter here");
9
10 // Say we wish to retrieve values
11 // into this type of container
12 using container = std::list<long double>;
13
14 int main()
15 {
16    const tree u235("n-092_U_235.xml");
17    auto data = u235(
18
19       // Maybe we remember it’s "somethingSuite"
20       // but don’t recall exactly...
21       seed
22          /"(.*)Suite",
23
24       // Create child_t with match string "reactions"...
25       seed
26          /"reactions",
27
28       // Create child_t with match string "reaction"
29       // and a filter...
30       seed
31          /"reaction"
32          + [](auto &node)
33          { return node(label) == "U236 + photon"; },
34
35       // Create child_t with match string "crossSection"...
36       seed
37          /"crossSection",
38
39       // Create child_t with match string "XYs1d"
40       // and a filter...
41       seed
42          /"XYs1d"
43          + [](auto &node)
44          { return node(label) == "recon"; },
45
46       // Start with values<> (not seed) to retain its
47       // not-entirely-trivial converter that extracts
48       // plain character data in the XML <values> node:
49       container{}
50          / values<>
51
52       /*
53       // Or, start with seed if you’re feeling adventurous:
54       container{}
55          / seed
56          / "values"
57          / [](auto &node, auto &list)
58            {
59               std::istringstream iss(node(pcdata,text));
60               container::value_type val;
61               while (iss >> val)
62                  list.push_back(val);
63            }
64       */
65    );
66 }
```