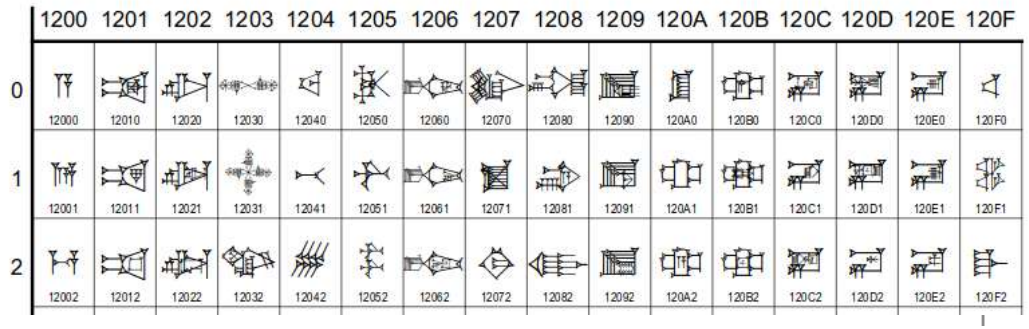
From CSV to Binary Data

Using DFDL and

12000 Cuneiform 120FF



License

Goals of this Training

- Learn how to self-teach about DFDL
  - What are the sources of information?
  - How to find things in the DFDL Spec
  - How structure a DFDL Schema project
    - setting it up for testing
    - composing schemas together
  - Where to get help

- Manipulate and learn DFDL schemas

- Learn enough DFDL properties to create an interesting and real DFDL Schema
  - We will build one, for NTP, on Day 3.

Day 1

- DFDL for CSV - deep dive - line-by-line review
  - XML Schema concepts - namespaces, targetNamespace, include/import, annotations
  - DFDL Top level formats, reusing a named format
  - Lookup and discuss each of the DFDL Properties
  - Run it from CLI (Lab 0)
  - Examine Tests built into the CSV schema
- CSV - change data to break it
  - Understanding diagnostics (Lab 1)
    - Schema Definition Error
    - Parse Error / Unparse Error
  - Improved Diagnostics (Lab 2)
  - Capture as a negative test case in TDML
  - Built-in-Self-Test (BIST)
  - TDML - a way of life when test is everything
  - Standard schema project layout
- CSV - evolve it in new directions (Start on Day 1)
  - Multiple delimiters (Lab 3)
    - Canonical form
    - Rount trip tests
  - Specific element names and types
    - dfdl:calendarPattern
  - Escape schemes (Lab 4)
  - Looking for DFDL Information
  - Runtime-valued delimiters (Lab 5)

Day 2

- CSV - evolve it in new directions (Finish on Day 2)
  - Multiple delimiters (Lab 3)
    - Canonical form
    - Rount trip tests
  - Specific element names and types
    - dfdl:calendarPattern
  - Escape schemes (Lab 4)
  - Looking for DFDL Information
  - Runtime-valued delimiters (Lab 5)
- Binary Data - 1
  - Alignment
  - Bit order, Byte order
  - Fill Byte
  - Optional Elements using Presence bits (Lab 6)
- Binary Data - 2
  - Unparsing
    - Computed elements (Lab 7)
  - hidden groups
  - Stored Length

Day 3

- Create a Real DFDL Schema: NTP
  - Starting from the spec
  - Example test data
  - Network Time Protocol
    - NTP (RFC 5905)
  - With TDML tests, etc.
  - Divide and Conquer as a Team
- Advanced Topics (If there is time)
  - Other lengthKinds
  - New things we're doing with DFDL - Unit normalization for VMF
  - Dealing with giant data format specs - spec scrapers.
- Wrap-up / Conclusions
  - Don't forget to provide feedback

Assumptions - Prerequisites

- Seen the DFDL Overview Presentation
- Know a bit of XML
  - w3schools XML Tutorial - basic introduction to XML.
  - Our Slides: Introduction to XML
- Know a bit of XML Schema (aka XSD)
  - w3schools XML Schema Tutorial - basics about XSD.
  - Our Slides: Introduction to XML Schema

CSV Deep Dive

CSV Deep Dive

- Line-by-line review
- XML Schema concepts - namespaces, prefixes, NCName and QName, targetNamespace, include/import, DFDL annotations
- DFDL Top level formats, reusing a named format
  - org/apache/daffodil/xsd/DFDLGeneralFormat.dfdl.xsd
  - Found in daffodil-lib module:
    - shortcut https://s.apache.org/daffodil-DFDLGeneralFormat.dfdl.xsd
- Lookup and discuss each of the DFDL Properties
- Run it from CLI
  - Doc Link: https://daffodil.apache.org/cli/
- Examine Tests built into the CSV schema

DFDL Core Concepts

- Infoset - a Data Model
  - DFDL Spec - Section 4 Figure 1
  - parse into the DFDL Infoset
  - unparse from the DFDL Infoset
  - NOT the same as the XML infoset
    - There is a mapping to/from XML and the DFDL Infoset
    - Specific to Apache Daffodil
    - see: https://daffodil.apache.org/infoset/

- Simple Types - subset of XSD/XML types
  - DFDL Spec - Section 5.1 Figure 3

TEST/QA for DFDL Schemas

Test Data Markup Language (TDML)

- XML-based language for writing (and managing) DFDL tests
  - parserTestCase
  - unparserTestCase
  - tests can do round-trips - parse [unparse [parse [unparse]]]
- A TDML file glues together
  - DFDL schema
  - test data (text, binary files, hex, bits)
    - input for parse, expected result for unparse
  - test infoset (XML)
    - input for unparse, expected result for parse
  - Can be in separate files (e.g., test.bin, test.xml, schema.dfdl.xsd, tests.tdml)
  - Can all be expressed directly in the TDML file itself (self-contained test in one TDML file)
    - Perfect for bug reports, or to get help/support with DFDL properties you don't understand
- Doc Link: https://daffodil.apache.org/tdml/
- XML Schema for TDML:
  - https://s.apache.org/daffodil-tdml.xsd

Standard File System Layout

- link: https://daffodil.apache.org/dfdl-layout/
- There are two "standard" layouts now
  - simplified layout - no namespaces. For small projects, learning
    - src
    - test
  - namespaced layout - supports packaging very large schemas composed of multiple projects
    - src/main/resources/myOrg/formatName
    - src/test/resources/myOrg/formatName
    - src/test/scala/myOrg/formatName
- We will use simple layout at first. Later use namespaced layout.
- A template system 'giter8' can be used to create an empty schema project
  - https://github.com/apache/daffodil-schema.g8

Built-In Self Test (BIST)

- Every DFDL Schema should have BIST
- Standard tool 'sbt' - Simple Build Tool
  - 'sbt test' - verifies schema works - loads all dependencies
    - including scala
    - including Daffodil and everything it depends on
    - including other schemas that this one uses (if they are published)
  - Runs suite of TDML tests (Test Data Markup Language)

CSV-Like Data

Lab 0

CSV - Change it, break it

- Modify data - add an extra field to a row or remove a field so the row is too short.
- The basic csv.dfdl.xsd schema tolerates this!
- Let's fix that.
  - Edit csvHeaderEnforced.dfdl.xsd schema so it does not accept this.
- Add a TDML negative test that ensures your schema detects this error in the data
  - Read about negative tests on the TDML doc page
- Add a Junit 1-liner so your test runs with 'sbt test'
  - src/test/scala/.....

CSV - Enhancing it

- As is, the CSV schema is pretty flawed
  - Fields all come through as "item" elements
  - All fields are string type despite DOB is always a date.
  - Can't have a comma inside a field - no escaping mechanism
- Let's start fixing these

Named & Typed Elements

Lab 1

NameDOB1.dfdl.xsd

- Replace "item" element with 4 local element declarations
  - 3 of these with appropriate names

    <xs:element name="...." type="xs:string" ... />
  - 1 of these

    <xs:element name="DOB" type="xs:date"

    ... date properties go here .... />
- Study tests in TDML file that use new schema
  - Has 1 negative test to be sure incorrect date syntax is caught

NameDOB1.dfdl.xsd

- "Left over data"
- Parse created an infoset that ignores the final faulty data
- This is *correct* behavior
- Parser back-tracks to end the record array when the parse of a record fails.
- So the parse succeeds. It just doesn't consume all the data.
- Next lab will modify the schema to get better diagnostics and reject faulty dates.

Different Kinds of Errors

- Schema Definition Error
  - the DFDL schema has an error
  - usually detected at schema compilation time (before parse/unparse begins)
  - sometimes detected at runtime
    - ex: if dfdl:lengthKind="delimited" dfdl:terminator="{ ../terminatorField }" but that expression returns "" (empty string).
- Parse Error
  - the data has an error or doesn't match the schema
  - causes backtracking to try other choice alternatives
  - causes optional elements/variable-length array elements to stop parsing more elements
  - only fatal if there are no alternatives for the parser to try
- Unparse Error
  - always fatal - unparsing fails
- Validation Error
  - if Daffodil is run with validation options selected
  - These do not cause backtracking
- Left-over data - warning (error if occurs in a TDML test)
  - parse succeeded, but did not consume all the data
  - This can be correct behavior if we are calling parse via API in a loop.
- TDML negative tests can expect any of these

Discriminators
More-Specific Diagnostics

Lab 2

DFDL discriminators

- Discriminators are used to "cut off possibilities"
- They discriminate a DFDL "point of uncertainty"
- Let's add one to nameDOB2.dfdl.xsd
  - A pattern discriminator takes a regex, and matches it against the data stream.

```
<dfdl:discriminator
  testKind="pattern"
  testPattern="."/>
```

  - Boilerplate: Must be wrapped in a sequence (so you can put it wherever you want)

```
<xs:sequence>
  <xs:annotation><xs:appinfo source='http://www.ogf.org/dfdl/'>
    <dfdl:discriminator testKind="pattern" testPattern="."/>
  </xs:appinfo></xs:annotation>
</xs:sequence>
```

NameDOB2.dfdl.xsd

- Add discriminator at start of record.
  - Suggest: Use a group definition and group reference to declutter.

```
<group name="discriminateAnyData">
 <sequence>
  <annotation><appinfo source="http://www.ogf.org/dfdl/">
   <dfdl:discriminator testKind="pattern" testPattern="."/>
  </appinfo></annotation>
 </sequence>
</group>
```

....to use the discriminator just...

```
<group ref="ex:discriminateAnyData"/>
```

- Do we get a more-specific error?
  - not "left over data"

Do we need dot-matches newline mode? (?s)

NameDOB2.dfdl.xsd

- Adding discriminator makes it possible to get more specific errors that mention the specific element and type
- discriminators provide format clarity about what deciding factor is that selects among alternatives
- discriminators can improve performance
  - for formats that do backtracking
  - backtracking aka "speculative parsing"

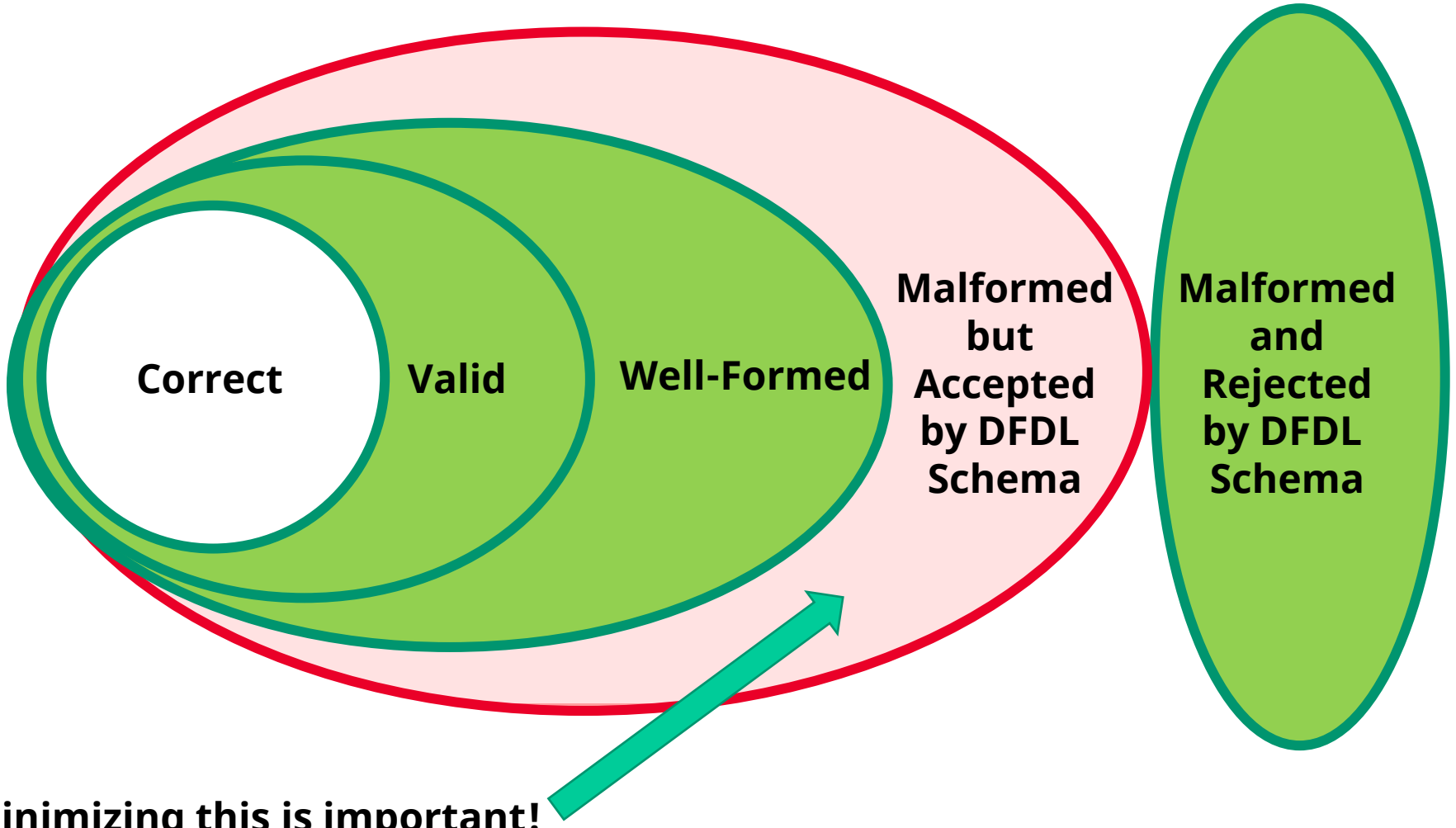
Negative Testing

Well-Formed vs. Valid vs. Correct

Quality Scale for Data

- Correct
  - Data that is perfect and suitable for all uses by intended applications
  - Blameless: If an application fails, that's its fault, not the fault of this data
- Valid
  - Data that satisfies "validity checks"
  - Establishes a policy about values in the data
  - Cares about values of numbers, patterns of text, co-existence constraints,.....
- Well-Formed
  - Data that has some value.
  - Applications may still want to use it even if it is not suitable for many things
  - "Worth talking about"
  - Cares that numbers are numbers, text is text, dates are dates
  - Can find and isolate all the pieces of the data
- Malformed
  - Data that shouldn't be considered.
  - Don't even want to bring it into memory
  - Might not even be what it says it is.
  - Dangerous data: Likely to crash applications - maybe even those trying to tolerate invalid data.

Don't Accept Malformed Data



Design to Exclude Malformed Data

- Schema should admit well-formed data
- Schema should exclude malformed data
  - And provide a good diagnostic.
  - True regardless of the fact that most data format documentation does not call out the diagnostic behavior.

- These goals are consistent with the DFDL schema being a good declarative specification of the format
  - Providing a good diagnostic makes it clearer what aspect of the specification is not being obeyed.

Well-Formed vs. Valid vs. Correct

- All these are a spectrum of how suitable is a given piece of data for the expected applications that consume it.
- DFDL schemas should be about parsing well-formed data, and rejecting malformed data.
  - Similarly they should be about unparsing well-formed infosets into well-formed data
- Sometimes constraints need to be expressed as part of well-formedness checking
  - DFDL Assertions may not be expressive enough.
  - Schematron rules could be used here
- But…this is still about well-formed data, not validity.

How to tell apart Well-Formed from Valid?

- Could you ever want the data available with this erroneous content in it?
    - For applications that want to tolerate some data mistakes?
    - For applications that want to help humans correct mistakes manually?
  - If so then you want that data to be considered well-formed, though invalid.
- Simple rule about Well-Formed
  - If it's not well-formed, you won't even get it into memory, so you can't touch it.
- DFDL Schemas can be designed to be strict or lax about what they accept.

Well-formed vs. Valid vs. Correct

- Test/QA needs can provide hints
  - Do you want to use your DFDL schema to generate erroneous data for test purposes?
  - If so by definition, that data will be well-formed according to your DFDL Schema
    - Because otherwise you can't use your DFDL Schema to generate it!
  - Such data will be Incorrect or Invalid, but Well-Formed.

Is it Well-Formed If.....?

- There is left-over data at the end of the file?
  - Maybe yes: if there is up to a few KBytes of it
    - A reasonable thing some file formats may allow
  - Clearly no: if there is 3 gigabytes of it.

- Sometimes it is a matter of degree!

Multiple Delimiters Canonical Form

Lab 3

Allowing Commas inside Comma-sep data

- Data formats use a variety of ways to fix this
  - Allow multiple terminators
    - Tab, | (aka pipe or vbar), or "//"
  - Escaping
  - Dynamic Terminator per row

Using Multiple Terminators: nameDOB3

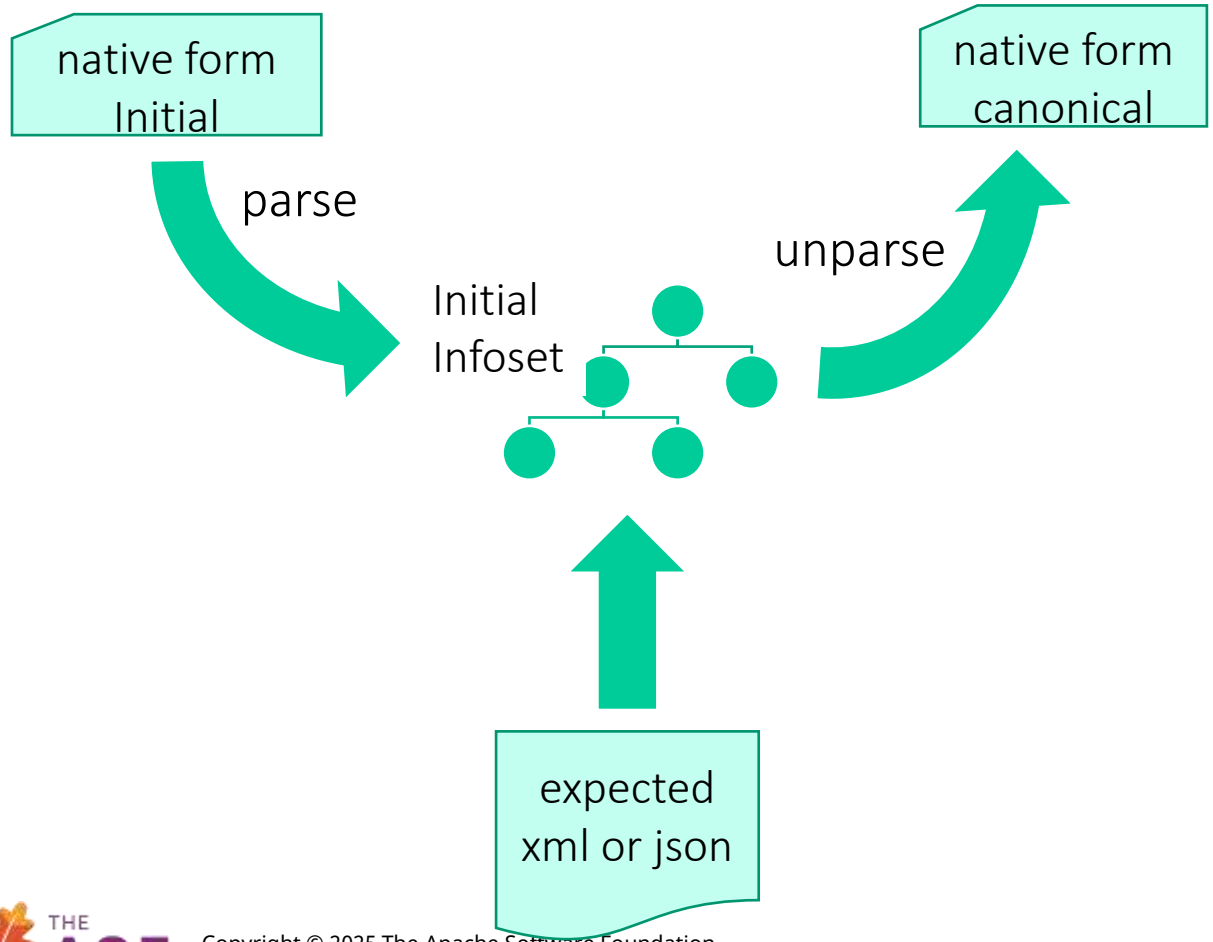
- Change dfdl:terminator="," to allow TAB, |, and // as terminators
  - Look at DFDL spec's description of terminator property
    - List of DFDL String Literals or DFDL Expression
    - XMLism - List means "whitespace separated list".
  - Lookup DFDL String Literal
    - A Tab is whitespace. So....use a DFDL Character Entity
- Study TDML Test that uses a mixture of terminators
- Issue: Unparse does NOT recreate the input data!
  - You get *Canonical Form* data out

Canonical Form is more Secure

- When formats offer alternatives *canonicalization* (c14n) improves data security
- Blocks covert channels
- Ex:
  - Format allows any amount of whitespace around comma-separators
  - Transmit covert data via number of spaces before/after the commas
  - Canonical form " , " (one space either side of comma) blocks the channel
- Insisting that data output is bit-for-bit identical is a holdover from inspect-only pass/fail data security

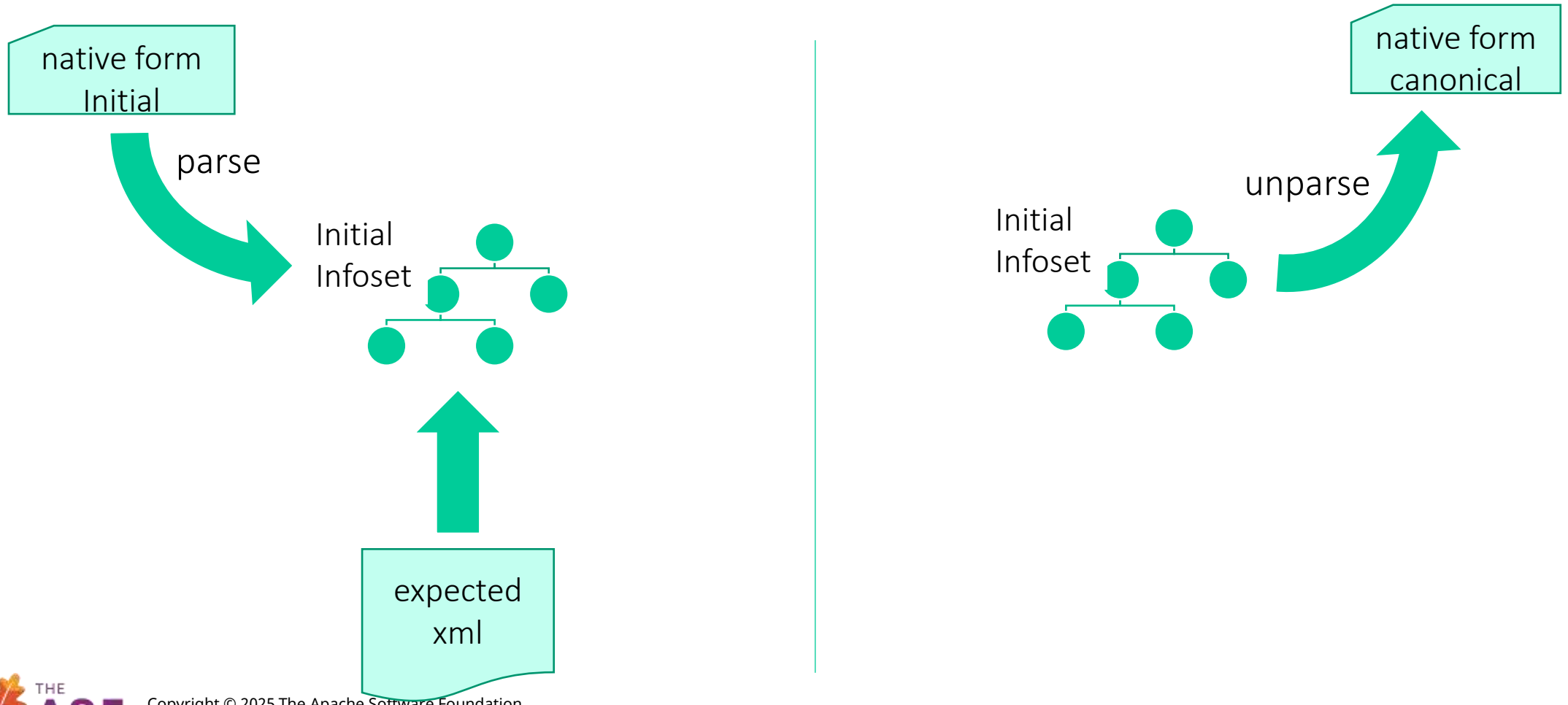
TDML Round Trip Parse/Unparse

- By default TDML tests run in roundTrip = "onePass" mode



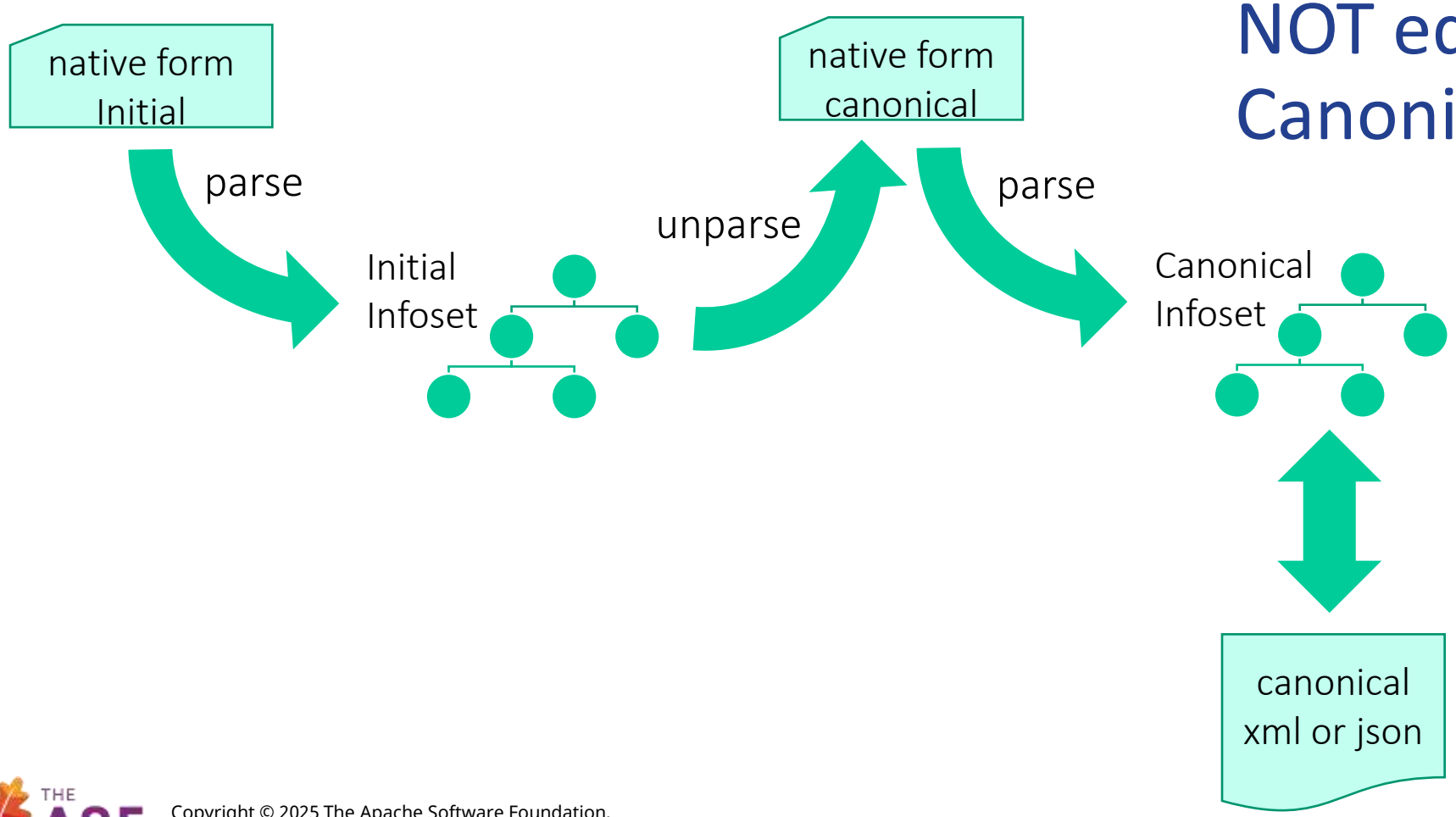
TDML Round Trip Parse/Unparse

- roundTrip="none"



TDML Round Trip Parse/Unparse

- roundTrip="twoPass"



- native form initial must NOT equal native form Canonical

Escape Schemes

Lab 4

Using Escape Schemes: nameDOB4

- DFDL has two kinds
  - escapeKind='escapeCharacter'
  - escapeKind='escapeBlock' (Let's use this one!)
- Lookup escape schemes in DFDL spec.
- Must add a top-level named escape scheme definition
  - Lookup defineEscapeScheme
- Must use it from the top-level default format via
  - escapeSchemeRef="..."
- Check out property dfdl:generateEscapeBlock
- XMLisms - how to embed " (double quote) into an XSD string literal?
  - escapeBlockStart='"' (that's single quote, double quote, single quote)
  - XML allows a string literal to start with single quotes or double quotes. Endings must match.
  - Or you could do escapeBlockStart="""

Dynamic Delimiter

Lab 5

Dynamic Terminator: nameDOB5

- Each row specifies its field terminator in first character
- Add element named "term" as new first column.
- dfdl:length="1" dfdl:lengthKind="explicit" dfdl:lengthUnits="characters"
- New xs:sequence for the 4 'real' elements
  - dfdl:terminator='{ ./term }'

Dynamic Terminator Variations

- Make the dynamic terminator be NUL (ascii 0)
- Working with NUL in DFDL is tricky
- XMLisms
  - XML documents cannot contain NUL. No way, No how.
  - Not even as &#x0;
  - Really
- TDMLisms
  - So a TDML file with embedded example data cannot have a literal NUL in it.
  - Fix 1: external data file, and <tdml:documentPart type="file">
  - Fix 2:
    - <tdml:documentPart replaceDFDLEntities="true">... %NUL;...
    - Use DFDL character entity for NUL which is %NUL;
    - Or Use DFDL numeric character entity %#x0;
    - Note: these create characters, not bytes. In a multi-byte character set it would matter!
- Expected Infoset
  - If you have Unicode, contains strange box characters. Like: □ So why?
  - See https://daffodil.apache.org/infoset/#xml-illegal-characters

dfdl:lengthKind and dfdl:lengthUnits

- Used frequently
  - delimited - what we've been using. Usually for text.
  - implicit
    - complex - length is sum of length of all children
    - simple - length depends on type (for binary data)
  - explicit - a constant or expression gives length
    - needs dfdl:lengthUnits
- Used in special cases
  - prefixed
    - needs dfdl:lengthUnits
  - pattern - uses regular expressions
  - endOfParent - not implemented (2022-06) by Daffodil
    - See https://daffodil.apache.org/unsupported/

Binary Data
Optional Elements with Flags
Packed Decimal

Lab 6

Binary Data Concepts

- Alignment, dfdl:alignmentUnits
- Mandatory Text Alignment
  - when text begins, we move to a boundary defined by the charset encoding.
  - Usually 8 bit boundary.
  - For 7-bit and smaller charsets no mandatory alignment (1-bit)
- dfdl:byteOrder
  - 'bigEndian' or 'littleEndian'
- dfdl:bitOrder
  - 'mostSignificantBitFirst' or 'leastSignificantBitFirst'
  - Not really order of the bits. Really just bit numbering scheme.

Binary Data Concepts

- dfdl:fillByte
  - Used to fill in unused space
    - DFDL Terminology:
      - "Padding" is about text
      - "Fill" is about binary
      - Lots of data formats use these terms in their own way however.
  - Commonly dfdl:fillByte="%#r00;" (zero byte)
    - %#rHH; notation is a DFDL Byte Entity aka a "raw byte".
  - Useful for debugging dfdl:fillByte="%#rFF;" (all 1's)
    - Filled data will show up in data more visibly.

Bit Order + Byte Order

- Most Significant Bit First + Big Endian
- Use Left-to-Right numbering to best visualize
- Ex: Integer of 24 bits not byte aligned
- Starts at bit 6 of byte 1

Byte: 1 2 3 4

00000001 10111010 01001111 10100000

xxxxx3 7 4 9 F 4

Bit Order + Byte Order

- Least Significant Bit First + Little Endian
- Use *Right-to-Left* numbering to best visualize
- Ex: Integer of 24 bits not byte aligned
- Starts at bit 4 of byte 1

Byte: 4 3 2 1

00000001 10111010 01001111 10100000

3 7 4 9 F 4 xxx

More on Bit Order

- See: https://daffodil.apache.org/tutorials/bitorder.tutorial.tdml.xml

TDML Data via Bits and Bytes

- You can create binary data directly in TDML files
- Often needed to construct detailed tests

```
<document>
  <documentPart type="byte">01BA 4FA0</documentPart>
  <documentPart type="bits">
    00000001 10111010 01001111 10100000
  </documentPart>
</document>
```

- R-to-L order and LSBF are supported also

Binary Data 1: nameDOB6

- Turn our CSV data into binary data
  - Make all elements optional
  - 4 single-bit flags at start of each record indicate presence of corresponding element
  - DOB date - stored as packed decimal

Binary Data 1: nameDOB6

- Separate flag and data creates a new situation
- What would happen if we unparse with a flag and data in inconsistent state?

<lastNamePI>0</lastNamePI>

....

<lastName>smith, jr.</lastName>

Binary Data
Hidden Groups
Output Value Calc

Lab 7

Binary Data 2: nameDOB7

- Put flags into a hidden group - not part of the Infoset
- Compute flags at unparse-time with dfdl:outputValueCalc
  - based on fn:exists(../lastName)

- This provides STRONG separation of format considerations from application logic.
- Application logic doesn't have to know the representation or that the format even has presence indicator flags

- This is an innovation in DFDL - no prior-gen format description language has this.
  - Everything else in DFDL is just standardizing prior practice.
  - To date, only Apache Daffodil implements this capability (not IBM DFDL yet)

Get REAL - DFDL Schema for NTP

Use Best Practices to Create a Real DFDL Schema

NTP - Network Time Protocol Messages

- Common Setup
- Review/Study *RFC 5905*
- Break into groups
- Create a repository on github per team
- Use sbt giter8 template
  - https://github.com/apache/daffodil-schema.g8
  - Follow README.md instructions
  - Create "professional" (namespaced = yes) layout schema
- TDML - capture test data bytes in the TDML file directly
- Bottom up - tests for sub-types in the schema

NTP "Schema Project"

- Use github/open-source SDLC
  - Use tickets for features and issues and coordinate activity across the team(s)
  - Each contributor creates a "fork" of the repository
  - Create "Pull-requests" to review and merge changes
    - Sometimes called "Merge requests".
- Best practices for DFDL
  - BIST - built in self test using TDML tests
    - contributions only accepted with tests showing they work
  - Shared types.dfdl.xsd file
  - LengthKind 'explicit' types use base simple type
  - New Daffodil Enums feature (extension to DFDL v1.0)
- Self-Contained TDML Test files
  - especially for unit tests of the types

Git/Github/Gitlab Best Practice

- Git allows many workflows - none is built in
- A project must choose and stick with a git workflow process

We suggest:

- Maintain a linear history - use *rebase*, not pull
  - makes it far simpler to isolate where bugs were introduced
- All changes done on forks
- Use branches named for issues/ticket numbers - allows work in parallel on many things
  - e.g., git checkout -b bug-NNN
- One feature or bug fix per PR
- Squash multiple commits of a single change/fix and its review cycle into a single commit before merging - avoids commits that are in inconsistent states
- Commit comments should specify *rationale* of changes. Explain *why.*
- Review all PRs
  - 2nd set of eyes required for any good SDLC
  - Call for specific reviewers if particular knowledge is needed
- Big sweeping changes - always do these as a separate change from any fix/functional changes.
  - file renaming, directory structure changes
  - whitespace/indentation standards change

- Setup automated continuous integration (CI) regression testing
  - Part of review is all CI tests must pass
  - Can copy from an existing DFDL schema (see github DFDLSchemas mil-std-2045 in the ".travis.yml" file)

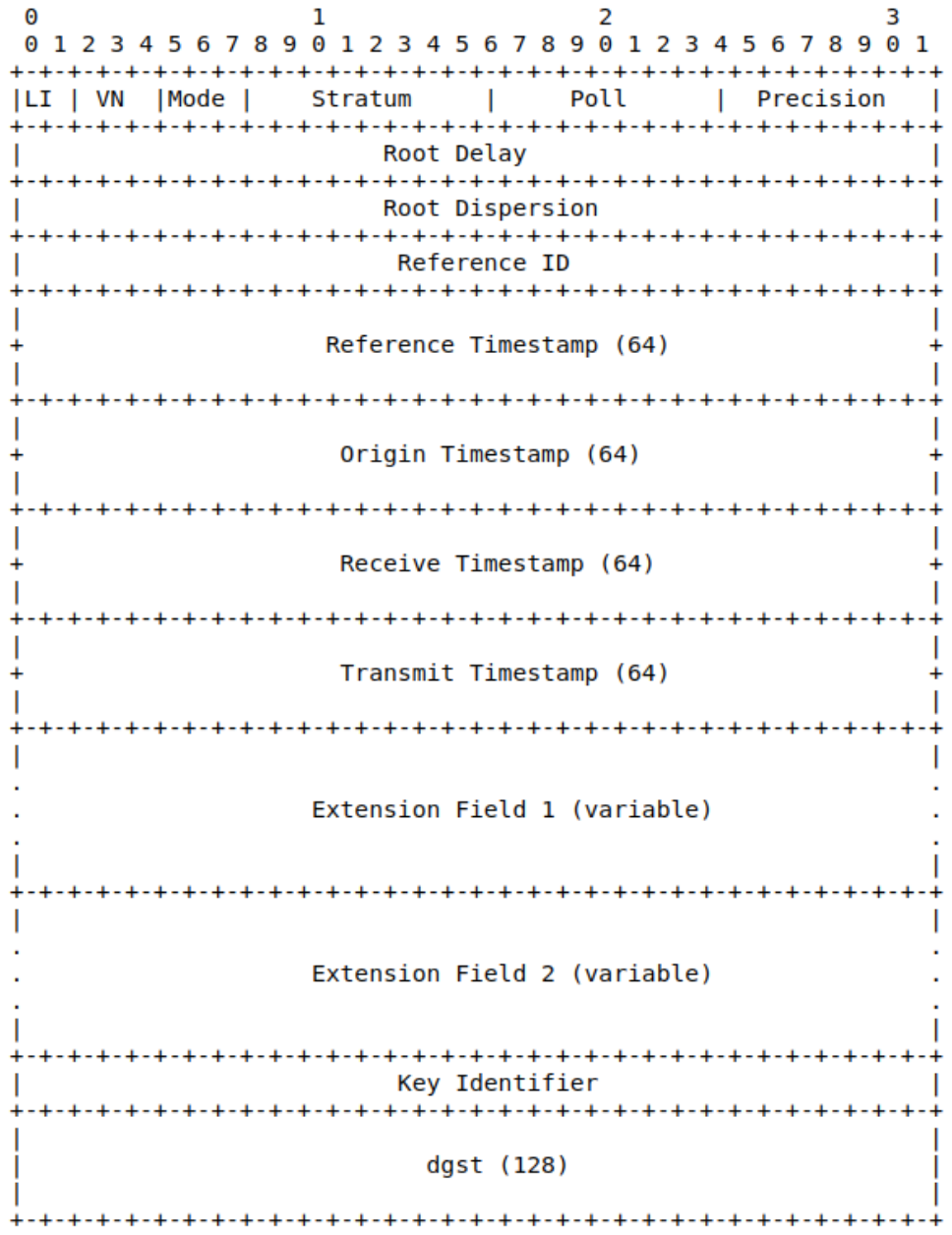
Git Cheat Sheet

- Using browser
  - https://github.com/OpenDFDL/dfdl-training-ntp-2022-07-28-team1.git
- Fork the repository (buttons upper right)
- git clone your fork to your local workstation via
  - git clone https://github.com/mbeckerle/dfdl-training-ntp-2022-07-28-team1.git
- or via ssh (saves typing passwords, but must setup public key at github.com profile)
  - git clone git@github.com:mbeckerle/dfdl-training-ntp-2022-07-28-team1.git
- Other command line git operations:
  - git checkout main # checkout main br
  - git checkout -b ntp-NNN-fix # create a fix branch and check it out
  - git add . # add your changes to a commit
  - git commit # commit your changes to the branch
  - git fetch --prune origin # pull down updates by others
  - git rebase origin/main # re-create your changes on top of them
  - git rebase -i origin/main # rebase interactive (for squashing fixup commits together)
  - git push origin ntp-NNN-fix # push your branch's commits (changes) for others to see
- Using browser: Create a pull request for others to review

Git Workflow

1. Do all your work locally, push to your fork repo
2. Name branches based on bug/issue numbers
3. Create PR (Pull Request) to merge to main in central repository
4. Request review
5. If changes are requested, fix, push again (to your fork)
   - Your changes will be added to the PR for re-review.
6. When your changes pass review...
   - squash changes into a single commit
   - rebase on top of any subsequent changes from others
   - retest to make sure it still works
   - push (with force) to your fork
     - git push --force origin myBranchName
   - merge (may require owner of main repo to do this)
7. Fetch from primary repo
8. Rebase your main onto the new main

NTP Packet Format



Enums

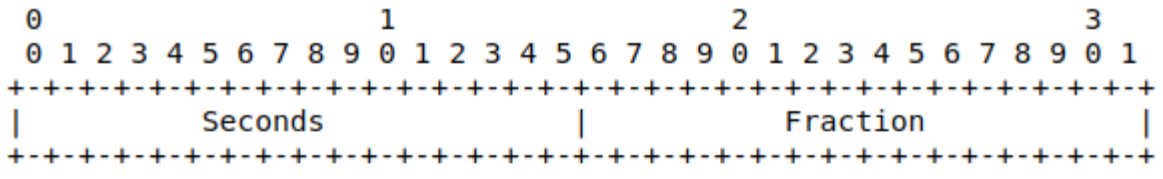
Stratum (stratum): 8-bit integer representing the stratum, with values defined in Figure 11.

Value	Meaning
0	unspecified or invalid
1	primary server (e.g., equipped with a GPS receiver)
2-15	secondary server (via NTP)
16	unsynchronized
17-255	reserved

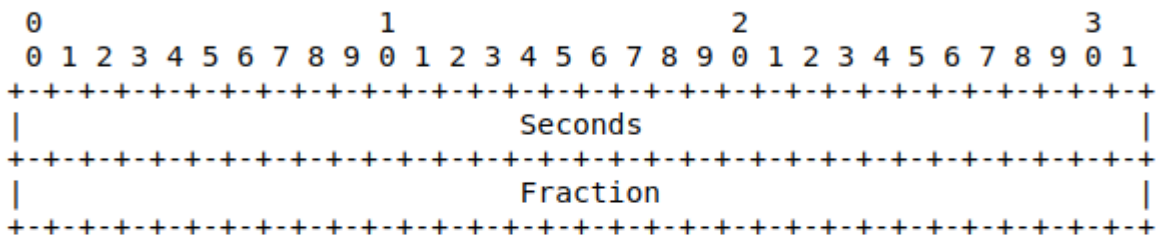
Figure 11: Packet Stratum

- NTP has many Enums
- Technique uses Daffodil Extensions to DFDL v1.0
- Copy it from mil-std-2045 schema
  - github DFDLSchemas mil-std-2045

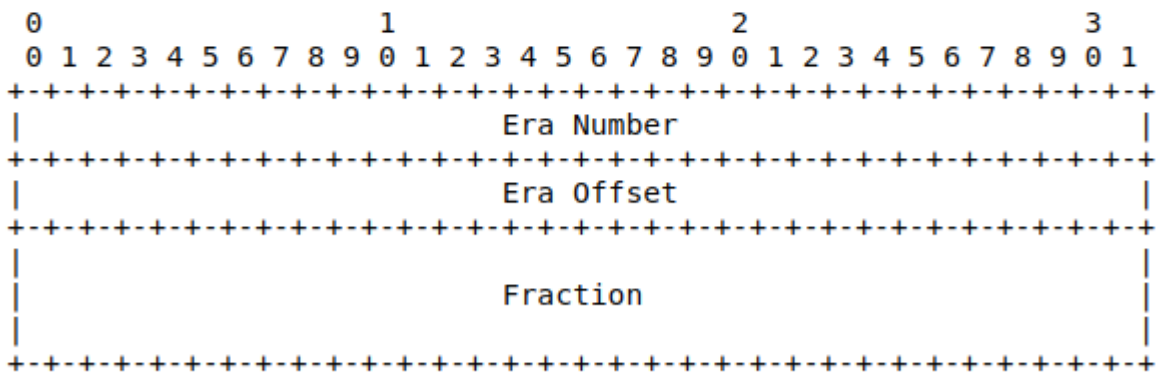
NTP Date/Times



NTP Short Format



NTP Timestamp Format



NTP Date Format

Figure 3: NTP Time Formats

dfdl:lengthKind 'implicit' vs. 'explicit'

- Complex type elements want dfdl:lengthKind 'implicit'
- Simple type elements want dfdl:lengthKind 'explicit'
- A whole schema file can only have one default
- Best Practice to avoid clutter/redundancy
  - Use lengthKind='implicit' as the default for all DFDL Schema Files
  - Create a types.dfdl.xsd schema file for all simple types
  - Create simple type base(s) for all simple types
- Base simpleType like this:

```
<simpleType name="UIntBase" dfdl:lengthKind='explicit'>
    <restriction base="xs:unsignedInt"/>
</simpleType>
```

- Every type that extends UIntBase will have explicit length:

```
<simpleType name="referenceID" dfdl:length="32">
  <restriction base="tns:UIntBase">
    <!-- if there are max/min facet constraints, they go here -->
  </restriction>
</simpleType>
```

Tasks

- Create a project: `sbt new apache/daffodil-schema.g8`
  - set namespaced option to yes
  - Main schema file will be pre-created.
- Create a types file - see next slide
- Create a simple type for all top level datatypes in an NTP packet.
  - At this point, all types can be simple unsigned integers with an appropriate length.
- Create a single type to be shared by all the timestamps.
- Create Enum types for the enumerated integers
- Using the previously created simple types, update the main schema to parse all Ntp Fields
- At the command line, try parsing some/all of the data files
- Update the Timestamp type to fully parse it
- Update the Root Delay and Root Dispersion types (these can be combined)
- Create and run test cases using the example data in TDML

Conclusion

Review: Goals of this Training

- Learn how to self-teach about DFDL
  - What are the sources of information?
  - How to find things in the DFDL Spec
  - How structure a DFDL Schema project
    - setting it up for testing
    - composing schemas together
  - Where to get help
- Manipulate and learn DFDL schemas
- Learn enough DFDL properties to create an interesting and real DFDL Schema
  - NTP

In Conclusion…

- Please provide feedback
  - Email to users@daffodil.apache.org

END

That's all folks.

Extra or draft slides may follow this slide.

Reject Elements

- Reject element means…
  - Part of the data didn't parse
  - We were able to determine how big it is
  - Create element as hexBinary

  - Ex: <unknown>090809afb9028ff</unknown>

- Should these be allowed?
  - Maybe yes: if there are a small number of reject records
    - A reasonable thing some file formats may allow
  - Clearly no: if there are no non-BLOB records. It's all BLOBs.

- Sometimes it is a matter of degree!

Reject Elements

- You want a reject element to be
  - well-formed
  - always invalid

- XSD Trick

```
<element name="unknown">
  <simpleType>
    <restriction base="xs:hexBinary"
      <maxLength value="0"/> <!-- always invalid -->
    </restriction>
  </simpleType>
</element>
```

Reject Elements

- Best to leave it up to the application
- Control from outside the DFDL Schema via externally set DFDL variable.
- Sometimes unavoidable - errors deep in the nest of data for a large file
  - that applications might be able to tolerate/skip.

Filtering Structured Text

- Data in this *CSV variant* format
- But Guard is XML-only.... ?

```
/foo/bar/data.csv

FIELD1, FIELD2, FIELD3
1, 2, [11,22,33]
4, sym_data, [66, 77]
/a/b/c, 9, 9873AF897FED080989873AF897FED080989873AF897FED0809898
```

Wrong! - Just a bypass

```
<? xml version="1.0" ?>
<textOK><![CDATA[
/foo/bar/data.csv

FIELD1, FIELD2, FIELD3
1, 2, [11,22,33]
4, sym_data, [66, 77]
/a/b/c, 9, 873AF897FED080989873AF897FED080989873AF897FED0809898
]]></textOK>
```

- This is technically valid XML for a trivial schema

```
<xs:element name="textOK" type="xs:string"/>
```

- Not in the spirit of XML for data verification, inspection, and sanitization.

Right - Parse Verifies Well-Formed

```
<d:csv1 xmlns:d="urn:com.tresys.dfdl/csv1">
  <version>1.0</version>
  <fileName>/foo/bar/data.csv</fileName>
  <columns>
    <column>FIELD1</COLUMN>
    <column>FIELD2</COLUMN>
    <column>FIELD3</COLUMN>
  </columns>
  <rows>
    <row>
      <c><i>1</i></c><c><i>2</i></c>
      <vector><v>11</v><v>22</v><v>33</v></vector>
    </row>
    <row>
      <c><i>4</i></c><c><s>sym_data</s></c>
      <vector><v>66</v><v>77</v></vector>
    </row>
    <row>
      <c><p>/a/b/c</p></c>
      <c><i>9</i></c>
      <hex>9873AF897FED080989873AF897FED080989873AF897FED0809898</hex>
    </row>
  </rows>
</d:csv1>
```


78

Is this CSV variant Well-Formed ?

DFDL Parse/Unparse can insure many things:

- Number of fields in each row matches the number of column headers.
- Only last column can be variable-length vector or hex blob.
- Fields can be tab or comma separated.
- Fields can have a maximum field length - excluding the vectors/blobs. (which could have a different max length)
- Fields syntax can either match the syntax of integers, identifiers, file names, dates/times, etc., for some list of acceptable field syntaxes.
- Hex blobs are hex-digits only. Enforce maximum length.
- Files obey a specified character-set encoding.
- Maximum number of rows/lines.
- Some characters are disallowed (control characters, for example).

Why is DFDL Needed? - ASN.1 ECN

- What about ASN.1 Encoding Control Notation?
- Already an ISO Standard (since 2008)
- Conceptually similar
  - Logical schema language + notations for physical representation
- Very different in the details.
- Developers [Love | Hate] [ASN.1 | XML]
- Differences that matter:
  - ASN.1 ECN
    - No open-source implementation (as of 2018-08-29)
    - Extension of a binary data standard ASN.1 BER/PER/DER
    - Goal to describe legacy protocol messages
  - DFDL
    - Open-source Daffodil implementation
    - Extension of a textual data standard XML
    - Goal to be union of data integration tool capabilities for format description

Things DFDL (v1.0 + BLOB) Does

- DFDL is for Images and Video
- Originally not in scope
- Large user demand to use DFDL on the metadata content of image file formats
  - Cybersecurity applications
- Adding BLOB (Binary Large Object) feature to DFDL language to enable DFDL to describe image files