Declarative Parsers in D

Ben Jones

DConf 2025

About Me

- Associate Professor, Lecturer at the University of Utah Kahlert School of Computing since 2017
- Lots of teaching in Masters of Software Development program (msd.utah.edu), along with many undergrad courses
- PhD from Utah in 2015 writing physics simulators for games/VFX
- Wrote C++ in grad school because everyone did
- Learned about new stuff in C++11, got excited about modern C++, ran across Andrei's "Iterators Must Go" talk and "The Case for D" etc


Caution
Not Tested In Prod

Main Challenge

ubyte[]

D Structs

Inspiration

- Pegged (https://github.com/dlang-community/Pegged/)
- Define a grammar, get a parser
- The Parser produces a parse tree which you can traverse
- Trying to learn about dmd, I was looking at the parse.d and AST nodes and wondering if PEGGED or similar could replace the dmd parser

```
import pegged.grammar;

mixin(grammar("
Arithmetic:
    Expr     <- Factor AddExpr*
    AddExpr  <- ('+'/'-') Factor
    Factor   <- Primary MulExpr*
    MulExpr  <- ('*'/'/') Primary
    Primary  <- '(' Expr ')'
              / Number
              / Variable
              / '-' Primary

    Number   <- [0-9]+
    Variable <- identifier
"));
```

Nobody wants a parse tree

- The Pegged generated parsers return parse trees, but I want an `Expression` or `FunctionDeclaration` or `TemplateInstance`
- So to convert dmd to using Pegged would mean doing 2 passes: one to build a parse tree, and a second to convert it into an AST
- It also means splitting parsing related code into to pieces:
  - The grammar rules
  - The Tree -> ASTNode conversion code
- Can we do it in one shot?

Enter Autoparsed

Autoparsed

- Structs and classes are annotated with their syntax (currently a lot less nice than Pegged's string based grammars)
- `auto parsed = parse!MyType(tokens);`
- Autoparsed builds a recursive descent parser based on their annotations
- Autoparsed provides helpers for quantifiers like ?, + and *

Using Autoparsed

```
auto s = parse!Statement(tokenStream)
alias Statement = OneOf!(AssignmentStatement, Expression);
@Syntax!(Identifier, eq, Expression)
struct AssignmentStatement {
  this(Identifier id, Expression exp){ ... }
}
```

- Autoparsed generates a recursive descent parser from the annotations and definitions
- The only parsing related part of `AssignmentExpression` is the `@Syntax` annotation
- `OneOf` syntax rules return a `SumType` on success

Where do we get a token stream?

```
@Token:
  enum lcurly = '{';
  enum rcurly = '}';
  enum lparen = '(';
  enum rparen = ')';
  enum comma = ',';
  enum semi = ';';
  enum eq = '=';
```

```
@Syntax!(RegexPlus!(OneOf!(' ', '\t',
'\n', '\r')))
struct Whitespace{
  const(char)[] val;
}

@Syntax!(RegexPlus!(OneOf!('-',
  InRange!('a','z'), InRange!('A', 'Z'))))
struct Identifier{
  const(char)[] val;
  alias val this;
}
```

Where do we get a token stream?

```
@Token:
  enum lcurly = '{';
  enum rcurly = '}';
  enum lparen = '(';
  enum rparen = ')';
  enum comma = ',';
  enum semi = ';';
  enum eq = '=';
```

```
@Syntax!(RegexPlus!(OneOf!(' ', '\t',
'\n', '\r')))
struct Whitespace{
  const(char)[] val;
}

@Syntax!(RegexPlus!(OneOf!('-',
  InRange!('a','z'), InRange!('A', 'Z'))))
struct Identifier{
  const(char)[] val;
  alias val this;
}
```

Where do we get a token stream?

```
@Token:
  enum lcurly = '{';
  enum rcurly = '}';
  enum lparen = '(';
  enum rparen = ')';
  enum comma = ',';
  enum semi = ';';
  enum eq = '=';
```

```
@Syntax!(RegexPlus!(OneOf!(' ', '\t',
'\n', '\r')))
struct Whitespace{
  const(char)[] val;
}
```

```
@Syntax!(RegexPlus!(OneOf!('-',
  InRange!('a','z'), InRange!('A', 'Z'))))
struct Identifier{
  const(char)[] val;
  alias val this;
}
```

Where do we get a token stream?

```
@Token:
  enum lcurly = '{';
  enum rcurly = '}';
  enum lparen = '(';
  enum rparen = ')';
  enum comma = ',';
  enum semi = ';';
  enum eq = '=';
```

```
@Syntax!(RegexPlus!(OneOf!(' ', '\t',
'\n', '\r')))
struct Whitespace{
  const(char)[] val;
}

@Syntax!(RegexPlus!(OneOf!('-',
  InRange!('a','z'), InRange!('A', 'Z'))))
struct Identifier{
  const(char)[] val;
  alias val this;
}
```

Lexer

- Lexer is a forward range with
  - `alias parseRule = OneOf!(tokenTypes!Module);`
  - `Nullable!(parseRule.NodeType) front_;`
  - `popFront()` calls `parse!parseRule(bytes_)` and does some error handling

Example Grammar: Simplified S-Expression

```
@Token{
  enum lparen = '(';
  enum rparen = ')';

  @Syntax!(RegexPlus!(OneOf!(' ', '\t', '\r', '\n')))
  struct Whitespace {
    const(dchar)[] val;
  }

  @Syntax!(RegexPlus!(Not!(OneOf!(lparen, rparen, Whitespace)), Token))
  struct Atom {
    const(dchar)[] val;
  }
}
```

S-Expression Grammar

```
@Syntax!(lparen, RegexPlus!(OneOf!(Atom, Sexp)), rparen)
class Sexp {
public:
  this(OneOf!(Atom, Sexp).NodeType[] members_){
    members = members_;
  }
private:
  OneOf!(Atom, Sexp).NodeType[] members;
}
```

Example Grammars

- S-expressions
- JSON
- Simplified C-Like language

Thoughts

- Implementation of `sexp` class is mostly independent of its grammar
- Aside from the `@Syntax` annotation, there's a small leak as the constructor takes `OneOf!(Atom, Sexp).NodeType[]` as a parameter
- It would be nice to return range instead of an array, but difficult because we need to empty the range before trying to parse anything else
- My implementation is a proof of concept that seems to work. Is the idea good?

Autoparsed vs DMD

- I sometimes look at the DMD codebase and try to find repeated patterns which can be eliminated with a good abstraction
- Basically every parse method in DMD is shaped like "look for these items in this order, and bail if one of them can't be parsed"
- In autoparsed, the logic for "look for these items in this order" is in one function
- Note: dumping on DMD is not my intention, and I'm not advocating for replacing it with autoparsed, but it provides a higher level of abstraction that has benefits (and costs)

Benefits of Grammar as Code

- If code is broken, it won't compile, so grammar issues are discovered right away
- Code can be introspected:

```
pragma(msg, "syntax rules for CLike grammar");
static foreach(i, sr; SyntaxRulesFromModule!clike){
  pragma(msg, "PEG string ", i, ": " ~ RuleToPegString!sr);
}
PEG string 0LU: clike.Whitespace <- RegexPlus(OneOf(` `, `           `, `
`))`
PEG string 1LU: clike.Identifier <- RegexPlus(OneOf(`-`, InRange(`a`, `z`), InRange(`A`, `Z`)))
PEG string 2LU: clike.Expression <- Identifier
...
PEG string 8LU: clike.IfStatement <- if_token `(` Expression `)` `{` OneOf(AssignmentStatement,
ExpressionStatement, IfStatement, WhileStatement) `}`
PEG string 9LU: clike.WhileStatement <- while_token `(` Expression `)` `{` OneOf(AssignmentStatement,
ExpressionStatement, IfStatement, WhileStatement) `}`
```

Grammar Parser Conflict in DMD

PrimaryExp `TypeCtor( Type )( ArgumentList)` is rejected when used alone in a ExpStatement #19833

Open

in https://dlang.org/spec/expression.html#primary_expressions this is the rule

```
  TypeCtor ( Type ) ( ArgumentListopt )
```

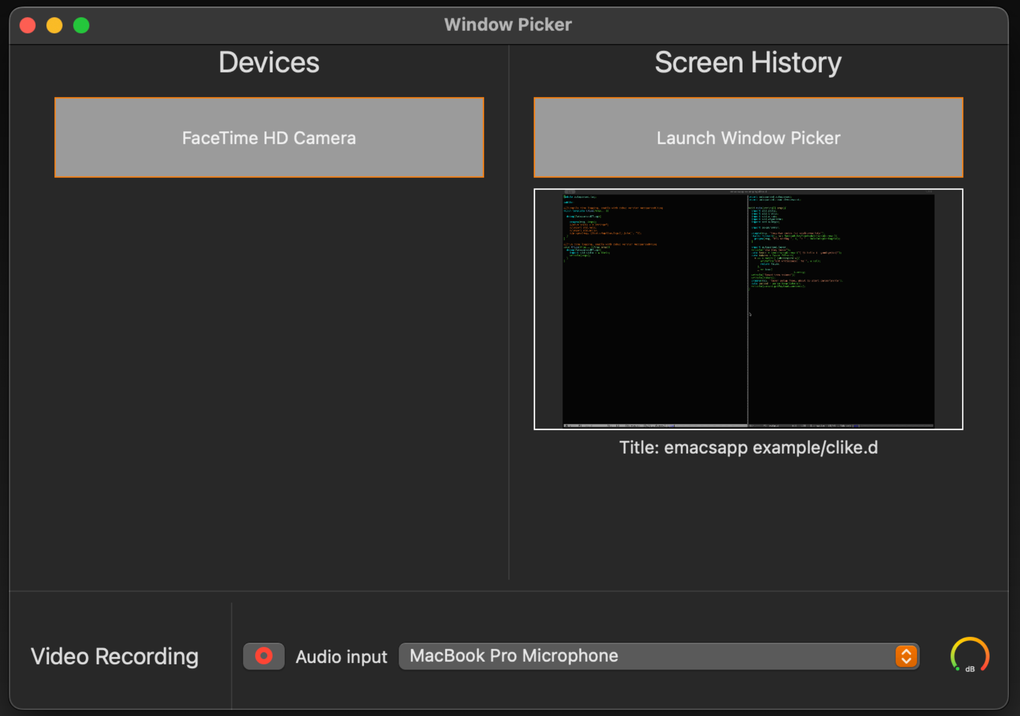
The parser takes the path of a declaration, it nevers tries a statement, in this case ExpressionStatement.

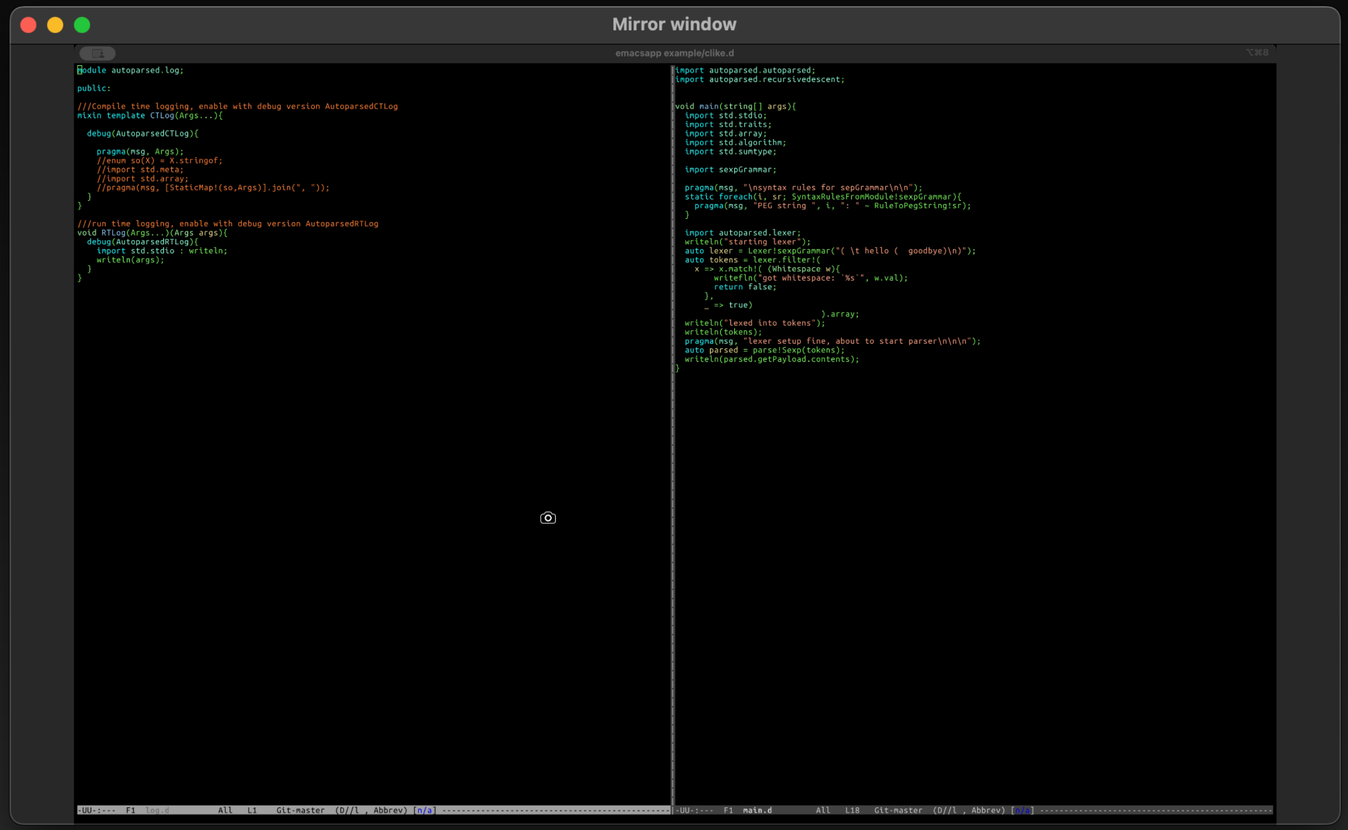
Almost Everything is "Quality of Implementation"

- Users of the library say almost nothing about how parsing should be performed
- We could also generate a LR parser (I think? I haven't done it) from the same annotations with (hopefully) no changes to our AST data types (maybe just `import autoparsed.~~recursivedescent~~lrparser`)
- Error handling and lots of other aspects could be adjust/improved by the library without users changing any of their own parsing code! Compare that to introduction of `errorSink` throughout DMD!

Part 2: MP4 files

PresenterMode



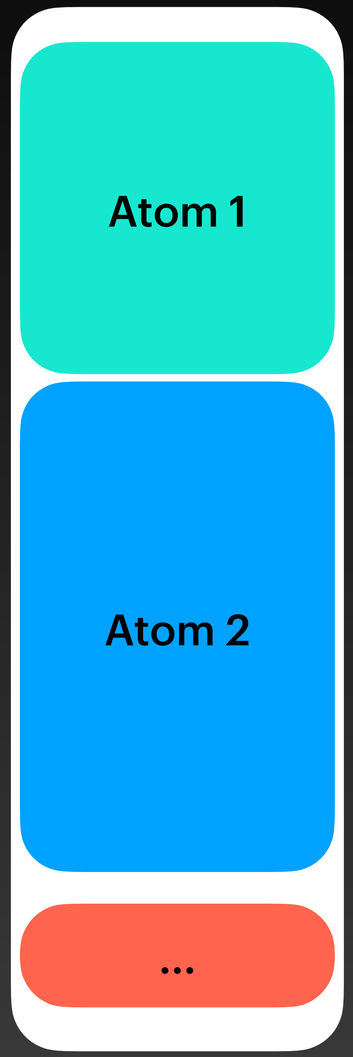


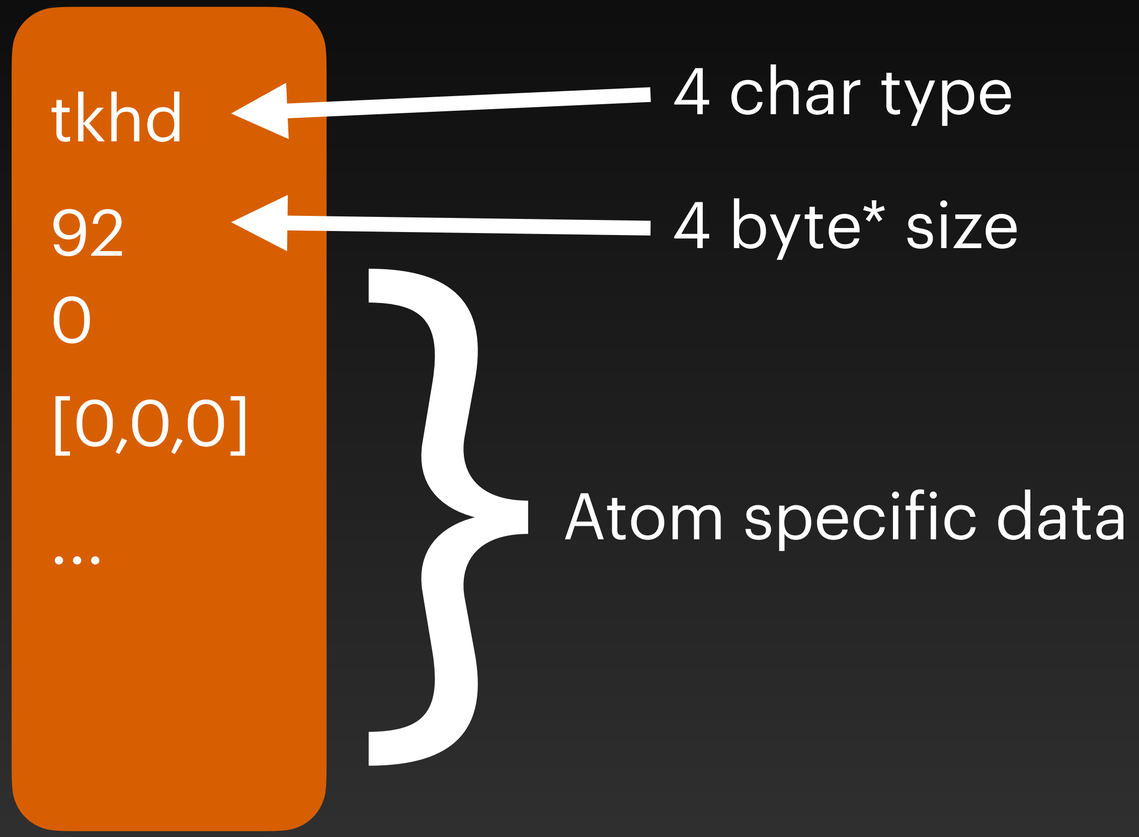
Laptop Screen

Projector Screen

https://github.com/benjones/presenterMode/

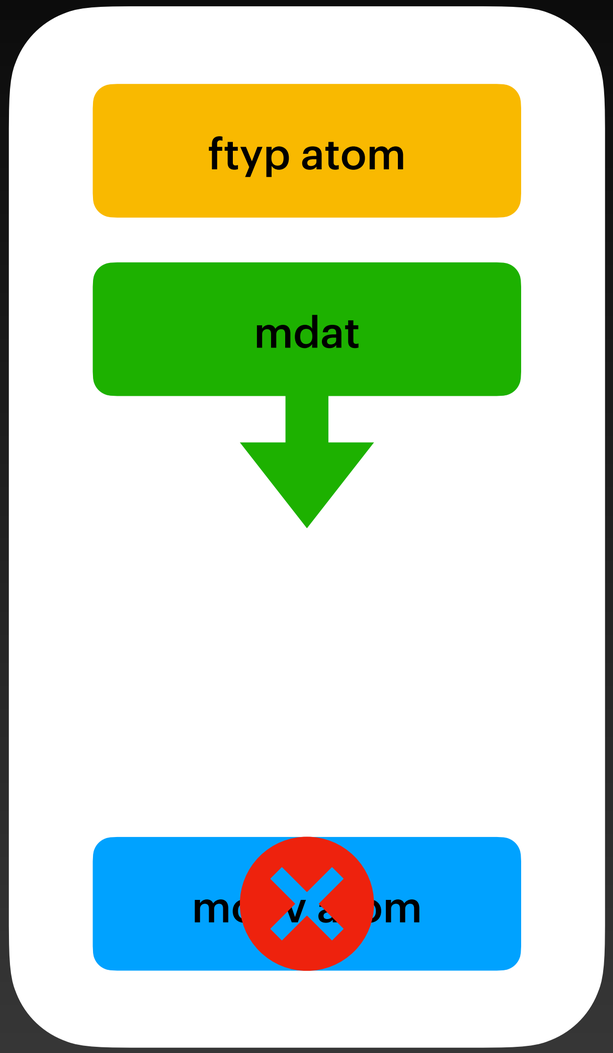
MP4 format



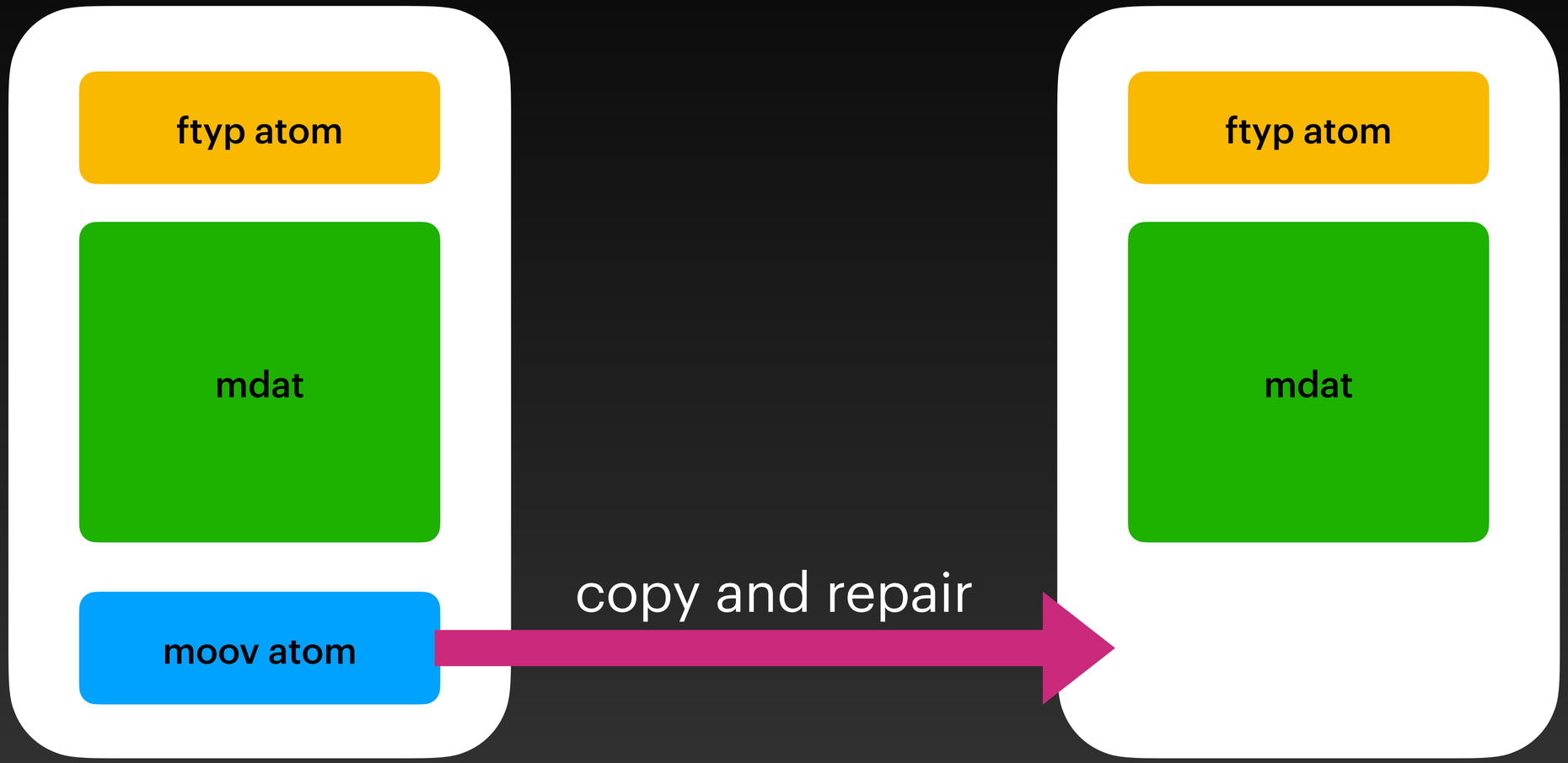


Sample Atom

Writing an MP4



Fixing an MP4



Valid MP4, same encoder

Broken MP4

MP4 Documentation

Overview

The track header atom specifies the characteristics of a single track within a movie. A track header atom contains a size field that specifies the number of bytes and a type field that indicates the format of the data (defined by the atom type `'tkhd'`).

The layout of a track header atom is as follows.

Data field	Bytes
Size	4
Type	4
Version	1
Flags	3
Creation time	4
Modification time	4
Track ID	4
Reserved	4
Duration	4
Reserved	8
Layer	2

• • •

Atom formats

- Atoms contain many fields, including other atoms
- Simple data is usually stored as big endian integers of a given size (sometimes fixed point numbers, sometimes byte arrays)
- There are 10s of relevant atoms for the files I'm looking at
- I started trying to come up with a good representation for an atom
  - `Tuple(name, size, representation)[]` ?

There is already a good D way of specifying how a set of fields are laid out:

`struct`

Structs!

- D structs do almost everything that I want except that their memory layout will not match the mp4 disk format
- If they did, I could just `mmap` the file and cast
- User code should just deal with structs which are mostly a dumb translation from the spec
- Library code should interact with the file with `mmap`
- Conversion should be transparent to user code
- This turned out to be really easy in D!

Main Ingredient: `opDispatch`

- For each field of the atom definition, I want to access it via `myAtom.myField` for both reading and writing
- This corresponds to two two `opDispatch` overloads
  - `fieldType opDispatch(string name)() if (name == "myField")`
  - `void opDispatch(string name)(FieldType val) if(name == "myField")`
- These overloads need to locate the correct offset and perform endian conversions

Details

- We can't use the native `offsetof` because D structs might be padded for alignment
- Pretty easy to compute with `std.traits` (`FieldNameTuple, Fields`)
- Some metaprogramming, but more CTFE than I expected originally
- `std.bitmanip` handles endianness stuff
- All implemented in `auto remapped(T)(ubyte[] data)` which returns a wrapper that mimics `T`
- Total implementation is < 300 lines, including extra features

User code

```
struct S1{
  int x;
  ulong y;
  ubyte[4] d;
}
ubyte[16] data;
data[3] = 1;
auto s1r = remapped!S1(data);
assert(s1r.x == 1);
s1r.x = 0x07060504;
```

```
S1 native = cast(S1)s1r;
assert(native.x == 0x07060504);

S1 other;
other.x = 1025;
other.y = 0x7777777777;
other.d = [8,9,10,11];
s1r = other;
assert(s1r.x == 1025);
```

User code

```
struct S1{
  int x;
  ulong y;
  ubyte[4] d;
}
ubyte[16] data;
data[3] = 1;
auto s1r = remapped!S1(data);
assert(s1r.x == 1);
s1r.x = 0x07060504;
```

```
S1 native = cast(S1)s1r;
assert(native.x == 0x07060504);

S1 other;
other.x = 1025;
other.y = 0x7777777777;
other.d = [8,9,10,11];
s1r = other;
assert(s1r.x == 1025);
```

User code

```
struct S1{
  int x;
  ulong y;
  ubyte[4] d;
}
ubyte[16] data;
data[3] = 1;
auto s1r = remapped!S1(data);
assert(s1r.x == 1);
s1r.x = 0x07060504;
```

```
S1 native = cast(S1)s1r;
assert(native.x == 0x07060504);

S1 other;
other.x = 1025;
other.y = 0x7777777777;
other.d = [8,9,10,11];
s1r = other;
assert(s1r.x == 1025);
```

User code

```
struct S1{
  int x;
  ulong y;
  ubyte[4] d;
}
ubyte[16] data;
data[3] = 1;
auto s1r = remapped!S1(data);
assert(s1r.x == 1);
s1r.x = 0x07060504;
```

```
S1 native = cast(S1)s1r;
assert(native.x == 0x07060504);
```

```
S1 other;
other.x = 1025;
other.y = 0x7777777777;
other.d = [8,9,10,11];
s1r = other;
assert(s1r.x == 1025);
```

User code

```
struct S1{
  int x;
  ulong y;
  ubyte[4] d;
}
ubyte[16] data;
data[3] = 1;
auto s1r = remapped!S1(data);
assert(s1r.x == 1);
s1r.x = 0x07060504;
S1 native = cast(S1)s1r;
assert(native.x == 0x07060504);

S1 other;
other.x = 1025;
other.y = 0x7777777777;
other.d = [8,9,10,11];
s1r = other;
assert(s1r.x == 1025);
```

Other features

- Arrays: `auto remapped(Layout: Layout[])(ubyte[] data)` returns a range
- Bitfields: annotate with the field names and sizes and generate `opDisptach` methods for each one

```
struct Packed {
  @(PackedField!("topNibble", 4),
    PackedField!("threeBits", 3),
    PackedField!("aBool", 1))
   ubyte packedByte;
}
```

```
ubyte[] bytes = [0xF2]; //1111_001_0
auto p = remapped!Packed(bytes);
assert(p.topNibble == 0xF);
assert(p.threeBits == 1);
assert(!p.aBool);
```

MP4 Specific Features

- `remapped` is a general purpose binary disk <-> d struct bridge
- D also helps with the specifics of MP4 structures
- Atom structs are annotated with `@NamedAtom("wxyz")`
- Using introspection, I can figure out which struct to remap the contents of an atom to based on its 4 char type
- There's annotations to handle atoms which contain atoms or arrays of other data inside them
- My 75 line traversal function can dump the atoms tree of a file, pretty printing each atom and all of its fields
- Each time I ran across a new item, I defined a `struct`, added an annotation, and it was dumped as expected after rebuilding

Example Atom

Overview

The track header atom specifies the characteristics of a single track within a movie. A track header atom contains a size field that specifies the number of bytes and a type field that indicates the format of the data (defined by the atom type `'tkhd'`).

The layout of a track header atom is as follows.

Data field	Bytes
Size	4
Type	4
Version	1
Flags	3
Creation time	4
Modification time	4
Track ID	4
Reserved	4
Duration	4
Reserved	8
Layer	2

```
@NamedAtom("tkhd")
struct TrackHeaderLayout {
    ubyte version_;
    ubyte[3] flags;
    uint creationTime;
    uint modificationTime;
    uint trackID;
    ubyte[4] reserved;
    uint duration;
    ubyte[8] reserved2;
    ushort layer;
    ...
}
```

Takeaways

- Structs and annotations are the most natural way to describe data
- D's metaprogramming features let programmers manipulate these descriptions to work with externally imposed layouts
- `getSymbolsByUDA` is an incredible tool for making anything declarative work

~~Pain Points~~ Opportunities for Improvement

- Bugs/errors/limitations in the code powering declarative systems can be extremely confusing for library users
- In particular, errors in `opDispatch` overloads lead to the overload being ignored, so the error message suggests that no overload exists
  - The errors suggest calling `opDispatch!"name"` directly which works, but feels like it shouldn't be necessary
  - I've gone out of my way to avoid `__traits(compiles)` for exactly this reason
  - Possible solution would be to add overloads which don't compile to the overload set. They can be ignored during resolution, but used for diagnostics

Template Lambdas

- Would make working with staticMap and Filter a bit nicer
- Not too onerous to create a named template, but would be nice to declare predicates/transforms at the point of use

`static foreach`

- Declaring variables in `static foreach` blocks leads to duplicate variable names
- You can use `{{` to create a new scope, but sometimes you don't want that either
- My go-to solution was just to rewrite common expressions over and over
- I "discovered" that a reasonable equivalent to "extract method" for metaprogramming are template mixins
- This initially didn't work, but I fixed Issue #21429

Reflections on Metaprogramming

- I **love** that lots of metaprogramming tasks are actually just normal D code run with CTFE (computing offsets, etc)
- When working with types, things start to give me C++ flashbacks
- Debugging a mess of templates can be really awful. No one wants a "stack trace" that mentions `StaticMap`
- I end up spraying `pragma(msg)` all over my code to track bugs down
- As a professor, I've seen students `printf` debug (and I'm not above that), but we can do better

A Compile Time Debugger

`pragma(breakpoint)`

`pragma(breakpoint)`

- Would pause the compiler during semantic processing and provide limited debugging features
  - backtrace -> show the current semantic analysis "stack"
  - print -> dump compiler AST nodes
  - continue -> return to compilation
- This seems implementable, with the biggest challenge being deciding what nodes to expose and the language for specifying it
- Would love to hear people's thoughts about this proposal

Final thoughts

- D's combination of features enables things that just aren't possible in other languages
- CTFE (awesome) and meta programming (powerful, but sometimes unwieldy) allow programmers to work magic at compile time
- High quality declarative libraries make user code succinct and difficult to write incorrectly
- Library authors need to take great care in hiding unintuitive and confusing errors from escaping
- We can do better at helping library authors

Thank you!

- https://users.cs.utah.edu/~benjones/
- https://github.com/benjones
  - Part 1 of the talk: autoparsed
  - Part 2: mp4fixer.remapped
  - Presenter Mode is the screen sharing tool