Saying NO to save a language

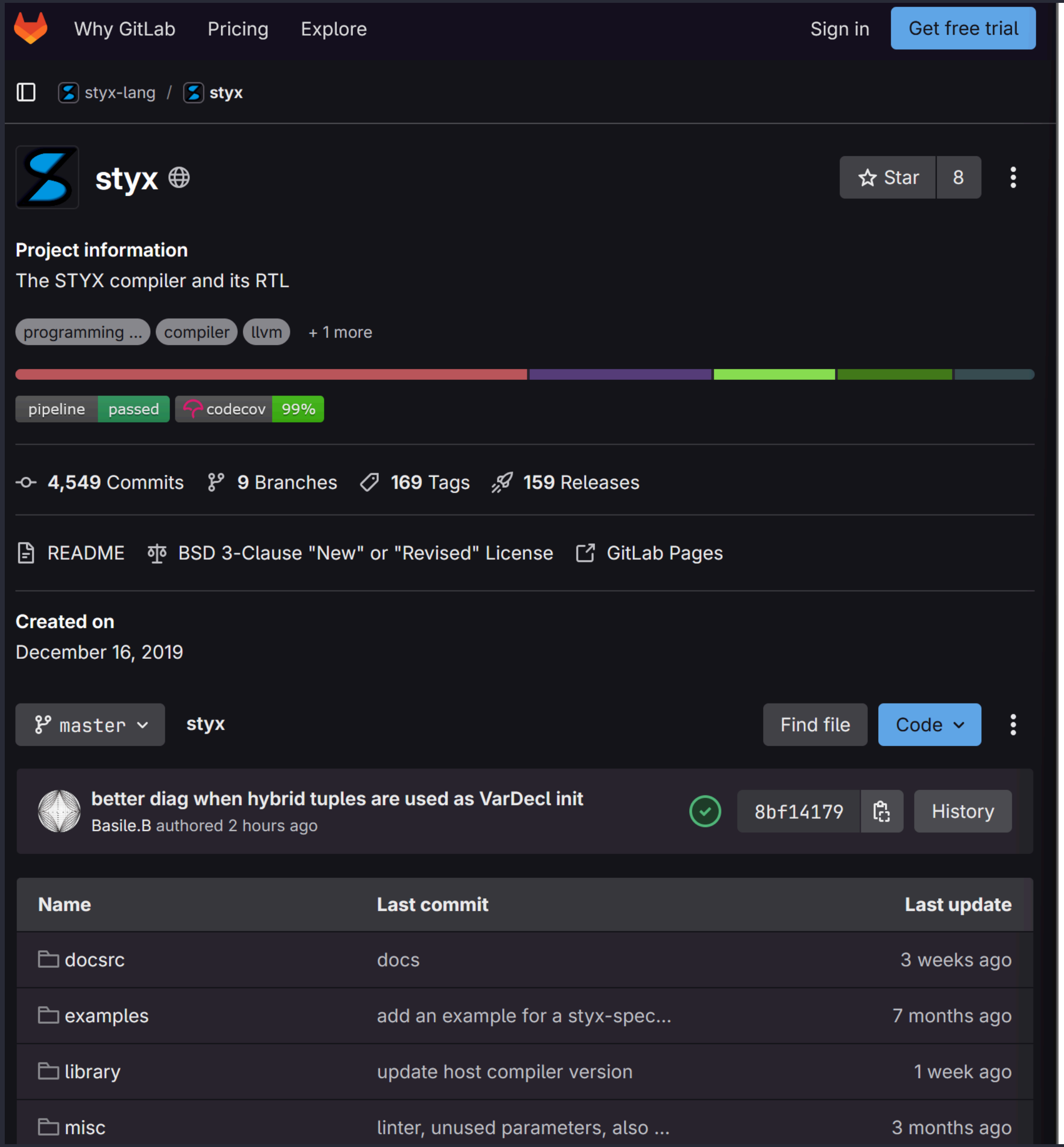
Why adding features to the compiler is so hard

Dennis Korpel

DConf'25 London - August 19 2024 · Slides: https://github.com/dkorpel/dconf

The circle of life

- C++ committee says "No" to Walter Bright's proposals
- Walter creates D
- D Improvement Proposals (DIPs) are made
- Walter says "No"
- D users create new languages



styx

Project information

The STYX compiler and its RTL

programming ... compiler llvm + 1 more

pipeline passed codecov 99%

4,549 Commits 9 Branches 169 Tags 159 Releases

README BSD 3-Clause "New" or "Revised" License GitLab Pages

Created on

December 16, 2019

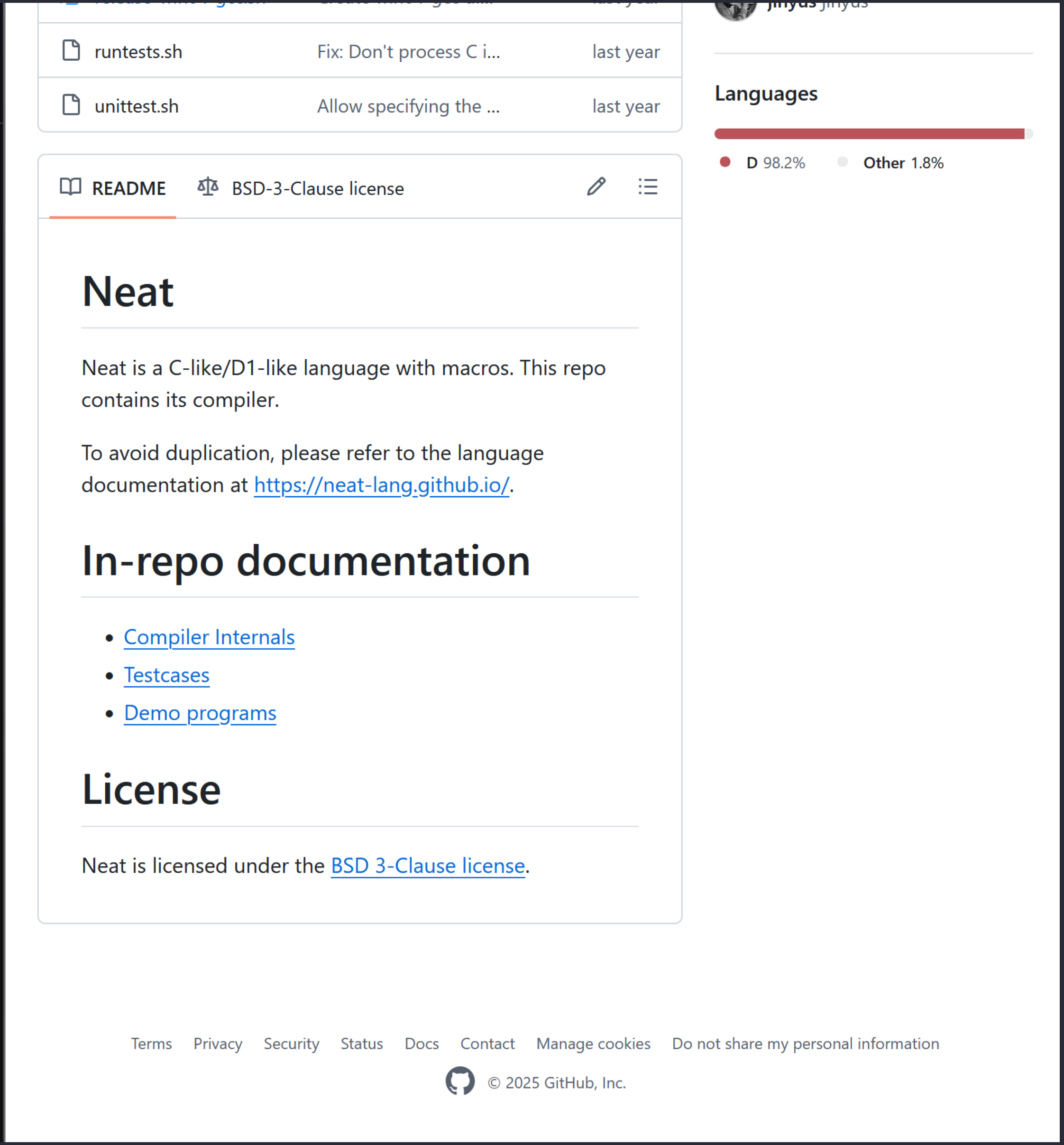
styx

better diag when hybrid tuples are used as VarDecl init

Basile.B authored 2 hours ago

8bf14179

Name	Last commit	Last update
docsrc	docs	3 weeks ago
examples	add an example for a styx-spec...	7 months ago
library	update host compiler version	1 week ago
misc	linter, unused parameters, also ...	3 months ago



runtests.sh	Fix: Don't process C i...	last year
unittest.sh	Allow specifying the ...	last year

Languages

● D 98.2% ● Other 1.8%

Neat

Neat is a C-like/D1-like language with macros. This repo contains its compiler.

To avoid duplication, please refer to the language documentation at https://neat-lang.github.io/.

In-repo documentation

- Compiler Internals
- Testcases
- Demo programs

License

Neat is licensed under the BSD 3-Clause license.

Styx: https://gitlab.com/styx-lang/styx · Neat: https://github.com/neat-lang/neat

Can't Walter just say yes?

Dr. No (1962) - United Artists · Yes Man (2008) - Warner Bros. Pictures

About me

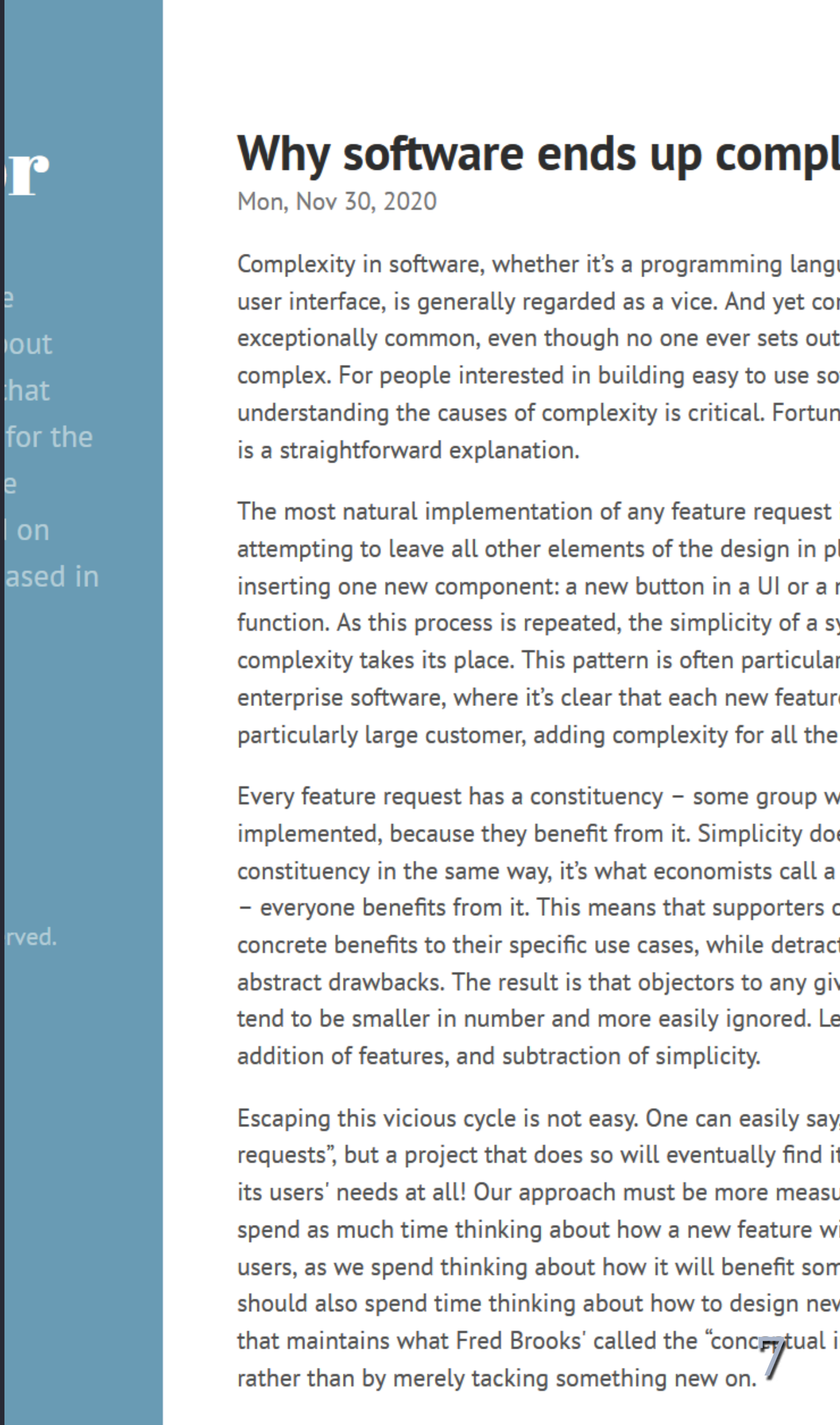
- Pull Request and Issue manager since 2022
- Want a small, stable programming language
- Also inclined to say

Contents

- How features add complexity
- How to make better improvement proposals
- How to refactor code to reduce technical debt

D is too complex

- Makes it harder to use/maintain
- How did it become this way?



r

e

out

hat

for the

e

l on

ased in

rved.

Why software ends up compl

Mon, Nov 30, 2020

Complexity in software, whether it's a programming langu
user interface, is generally regarded as a vice. And yet cor
exceptionally common, even though no one ever sets out
complex. For people interested in building easy to use so
understanding the causes of complexity is critical. Fortun
is a straightforward explanation.

The most natural implementation of any feature request
attempting to leave all other elements of the design in pl
inserting one new component: a new button in a UI or a n
function. As this process is repeated, the simplicity of a sy
complexity takes its place. This pattern is often particular
enterprise software, where it's clear that each new feature
particularly large customer, adding complexity for all the

Every feature request has a constituency – some group w
implemented, because they benefit from it. Simplicity doe
constituency in the same way, it's what economists call a
– everyone benefits from it. This means that supporters o
concrete benefits to their specific use cases, while detract
abstract drawbacks. The result is that objectors to any giv
tend to be smaller in number and more easily ignored. Le
addition of features, and subtraction of simplicity.

Escaping this vicious cycle is not easy. One can easily say
requests", but a project that does so will eventually find it
its users' needs at all! Our approach must be more measu
spend as much time thinking about how a new feature wi
users, as we spend thinking about how it will benefit som
should also spend time thinking about how to design nev
that maintains what Fred Brooks' called the "conceptual i
rather than by merely tacking something new on.

https://alexgaynor.net/2020/nov/30/why-software-ends-up-complex/

Simplicity gets dismissed

- Features are naturally additive
- Supporters claim concrete benefits
- Detractors claim abstract drawbacks
  - Sounds like exaggerating
- Add 1% a hundred times and you triple the size
  - (Exponential growth)

Not all compiler stages are equal

Parsing → Semantic Analysis → Code generation

- Parsing
  - ~10 KLOC in dmd, 'solved' problem
- Code generation
  - Outsourced to LLVM, GCC, or Walter Bright
  - ~100 KLOC in dmd
- Semantic analysis
  - ~200 KLOC, 'heart' of the D language

"Semantic" is the trouble spot

But what is it?

Pictured: ArcelorMittal Orbit

It's tree rewriting

```
x + y * 0
```

Tree form:

```
   +                +            x
  / \              / \
 x   *      →     x   0    →
    / \
   y   0
```

Just recursion and if-statements

```
Expression semantic(Expression exp)
{
    exp.lhs = semantic(exp.lhs);
    exp.rhs = semantic(exp.rhs);

    if (exp.kind == MULTIPLICATION && exp.rhs == Expression(0))
        return Expression(0);

    if (exp.kind == ADDITION && exp.rhs == Expression(0))
        return exp.lhs;
}
```

…Multiplied by 20000

Example of implementation woes

- Command to run unittests for single module:

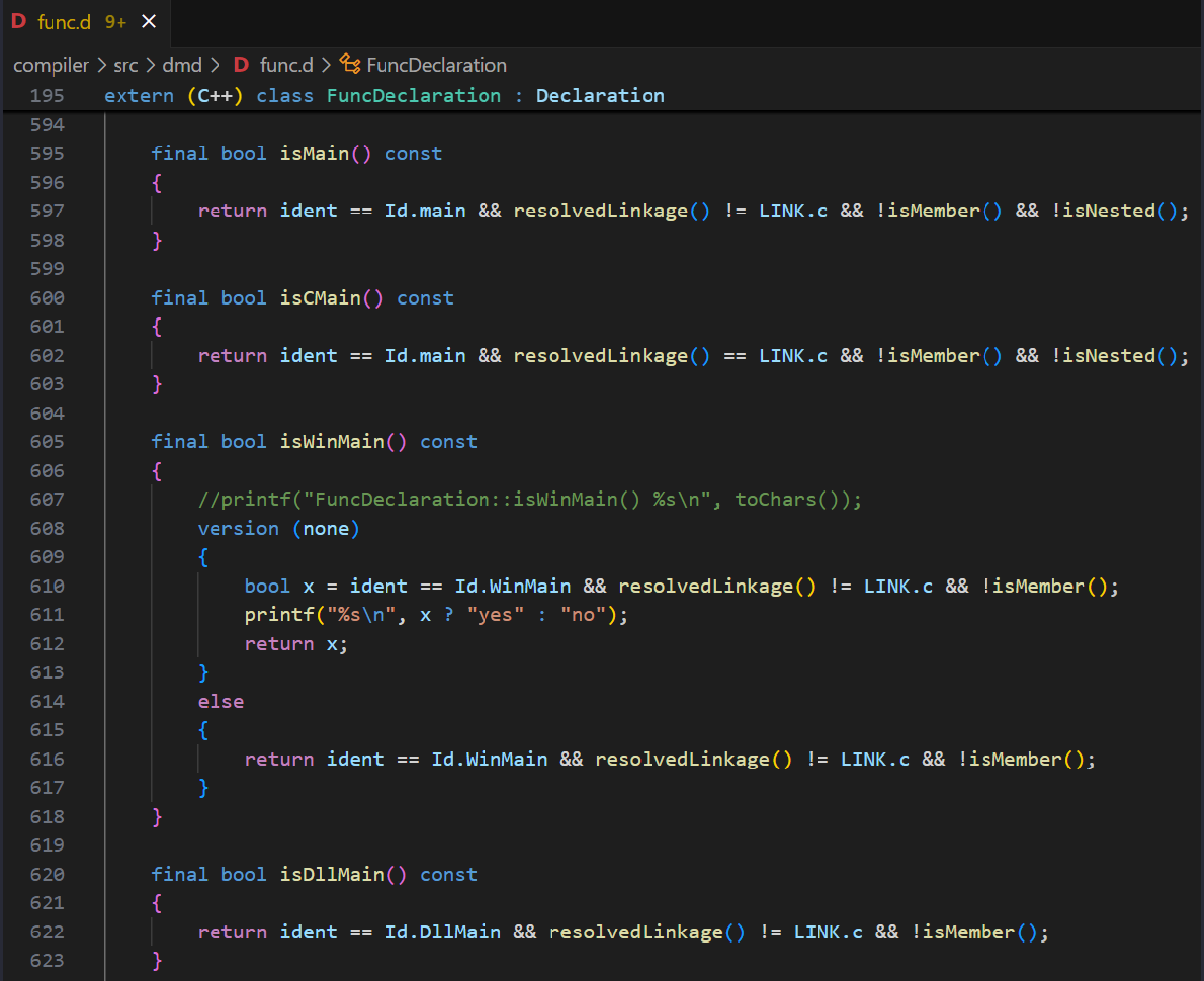
```
dmd -i -unittest -main -run foo.d
```

- `-unittest` only compiles in `unittest {}` functions
- `-main` implicitly adds `void main() {}`
- What if `foo.d` already has a `main`?
- `Error: only one main allowed`
- Enhancement request: only add empty main when needed

https://github.com/dlang/dmd/pull/13057

Contributions become harder

- First question: How to find existing main?
  - In C, this could be a simple check in the parser
  - In D, consider `mixin` `static if (X)` `import`
- Parsing is too early
- Check for main in code generator?
  - Too late, backend is separate from frontend
- Another question: *what is* `main` *?*



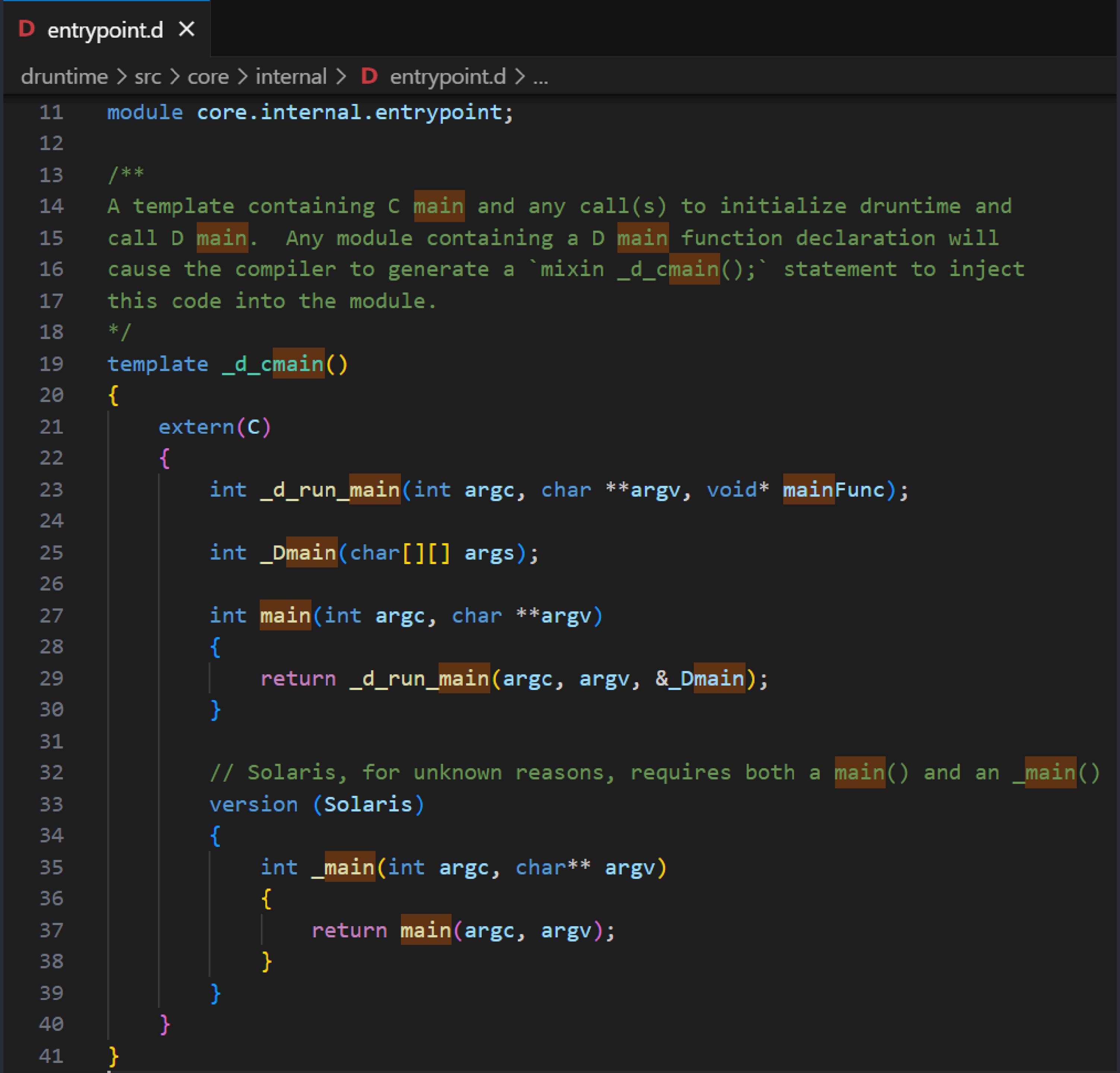
```
extern (C++) class FuncDeclaration : Declaration

    final bool isMain() const
    {
        return ident == Id.main && resolvedLinkage() != LINK.c && !isMember() && !isNested();
    }

    final bool isCMain() const
    {
        return ident == Id.main && resolvedLinkage() == LINK.c && !isMember() && !isNested();
    }

    final bool isWinMain() const
    {
        //printf("FuncDeclaration::isWinMain() %s\n", toChars());
        version (none)
        {
            bool x = ident == Id.WinMain && resolvedLinkage() != LINK.c && !isMember();
            printf("%s\n", x ? "yes" : "no");
            return x;
        }
        else
        {
            return ident == Id.WinMain && resolvedLinkage() != LINK.c && !isMember();
        }
    }

    final bool isDllMain() const
    {
        return ident == Id.DllMain && resolvedLinkage() != LINK.c && !isMember();
    }
```



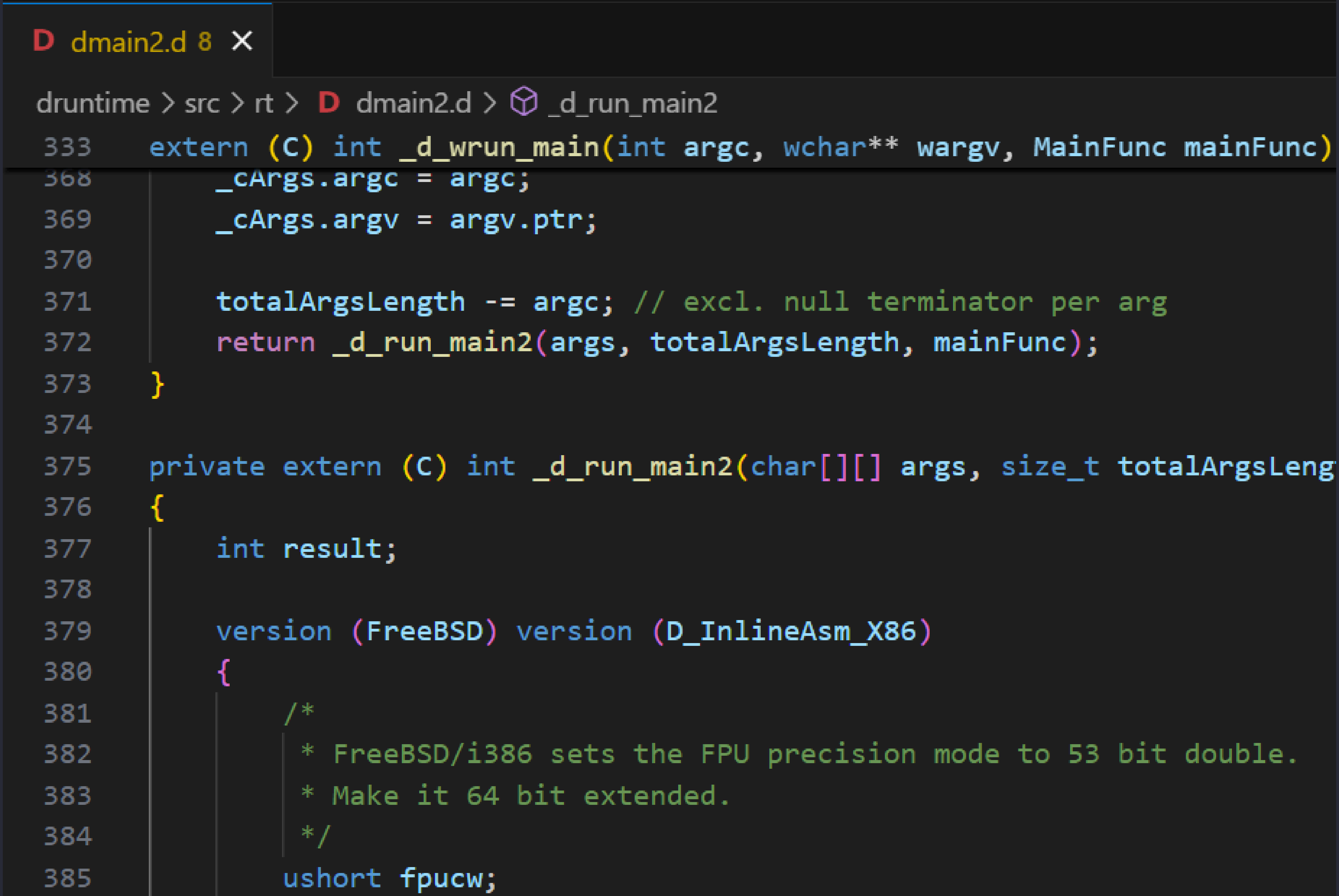
```d
module core.internal.entrypoint;

/**
A template containing C main and any call(s) to initialize druntime and
call D main.  Any module containing a D main function declaration will
cause the compiler to generate a `mixin _d_cmain();` statement to inject
this code into the module.
*/
template _d_cmain()
{
    extern(C)
    {
        int _d_run_main(int argc, char **argv, void* mainFunc);

        int _Dmain(char[][] args);

        int main(int argc, char **argv)
        {
            return _d_run_main(argc, argv, &_Dmain);
        }

        // Solaris, for unknown reasons, requires both a main() and an _main()
        version (Solaris)
        {
            int _main(int argc, char** argv)
            {
                return main(argc, argv);
            }
        }
    }
}
```



```
extern (C) int _d_wrun_main(int argc, wchar** wargv, MainFunc mainFunc)
    _cArgs.argc = argc;
    _cArgs.argv = argv.ptr;

    totalArgsLength -= argc; // excl. null terminator per arg
    return _d_run_main2(args, totalArgsLength, mainFunc);
}

private extern (C) int _d_run_main2(char[][] args, size_t totalArgsLeng
{
    int result;

    version (FreeBSD) version (D_InlineAsm_X86)
    {
        /*
         * FreeBSD/i386 sets the FPU precision mode to 53 bit double.
         * Make it 64 bit extended.
         */
        ushort fpucw;
```

Once it's in there, it stays

- Working on `final switch`-related code, I discovered:
    - switch case statement can be runtime `int` variable
    - `enum` can enumerate struct with `opBinary!"+"`
- Can we remove these please?
    - Breaks existing code

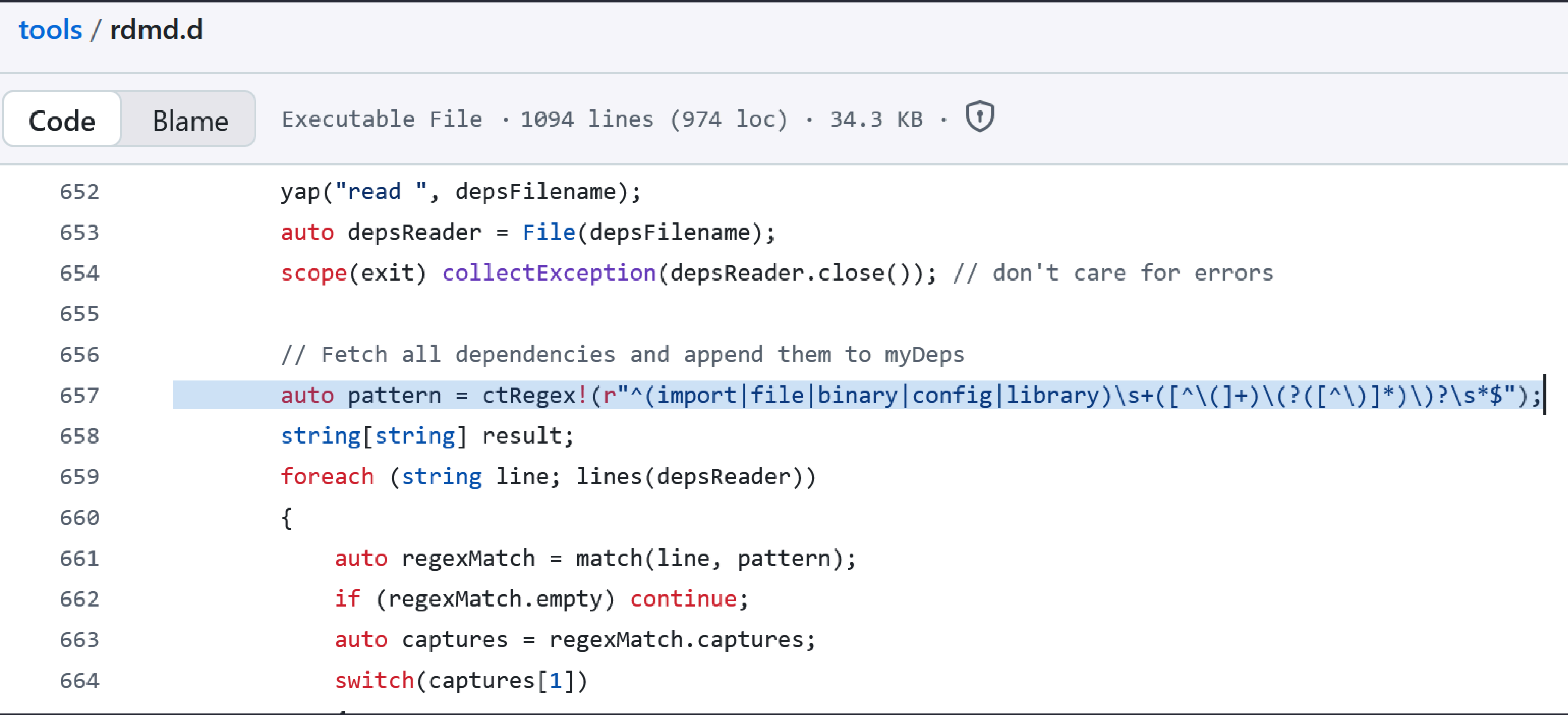
All behaviors are depended on

Hyrum’s Law:

> With a sufficient number of users of an API, it does not matter what you promise in the contract: All observable behaviors of your system will be depended on by somebody.

- D exposes compiler internals (`.stringof`, `.mangleof`, etc.)
- D users unittest those internals
- Even `dmd -v` verbose output depended on by `rdmd`

https://www.hyrumslaw.com/



tools / rdmd.d

Executable File · 1094 lines (974 loc) · 34.3 KB ·

```
        yap("read ", depsFilename);
        auto depsReader = File(depsFilename);
        scope(exit) collectException(depsReader.close()); // don't care for errors

        // Fetch all dependencies and append them to myDeps
        auto pattern = ctRegex!(r"^(import|file|binary|config|library)\s+([^\(]+)\(?([^\)]*)\)?\s*$");
        string[string] result;
        foreach (string line; lines(depsReader))
        {
            auto regexMatch = match(line, pattern);
            if (regexMatch.empty) continue;
            auto captures = regexMatch.captures;
            switch(captures[1])
```

https://github.com/dlang/tools/blob/4d4a2798b6f89befbdbc766b8284aae229c3c66b/rdmd.d#L657

How features add complexity (conclusion)

- We add more than we remove
- Compiler development becomes harder/slower
- But: 'never add any features' is not a solution either

> It's 2025, where are my tuples and sum types!

5 tips for improvement proposals

Pictured: Battersea power station

#1 - Include real usage examples

> Let's add magic `__REACHABLE__` boolean

- Why?
  - "For when you want to know whether code is reachable" 🤨
  - "Why not" ❌
  - Code example of usage in context ✅
  - GitHub link to production code that needs it ✅👍👏💯

#2 - Inspire errors by real bugs

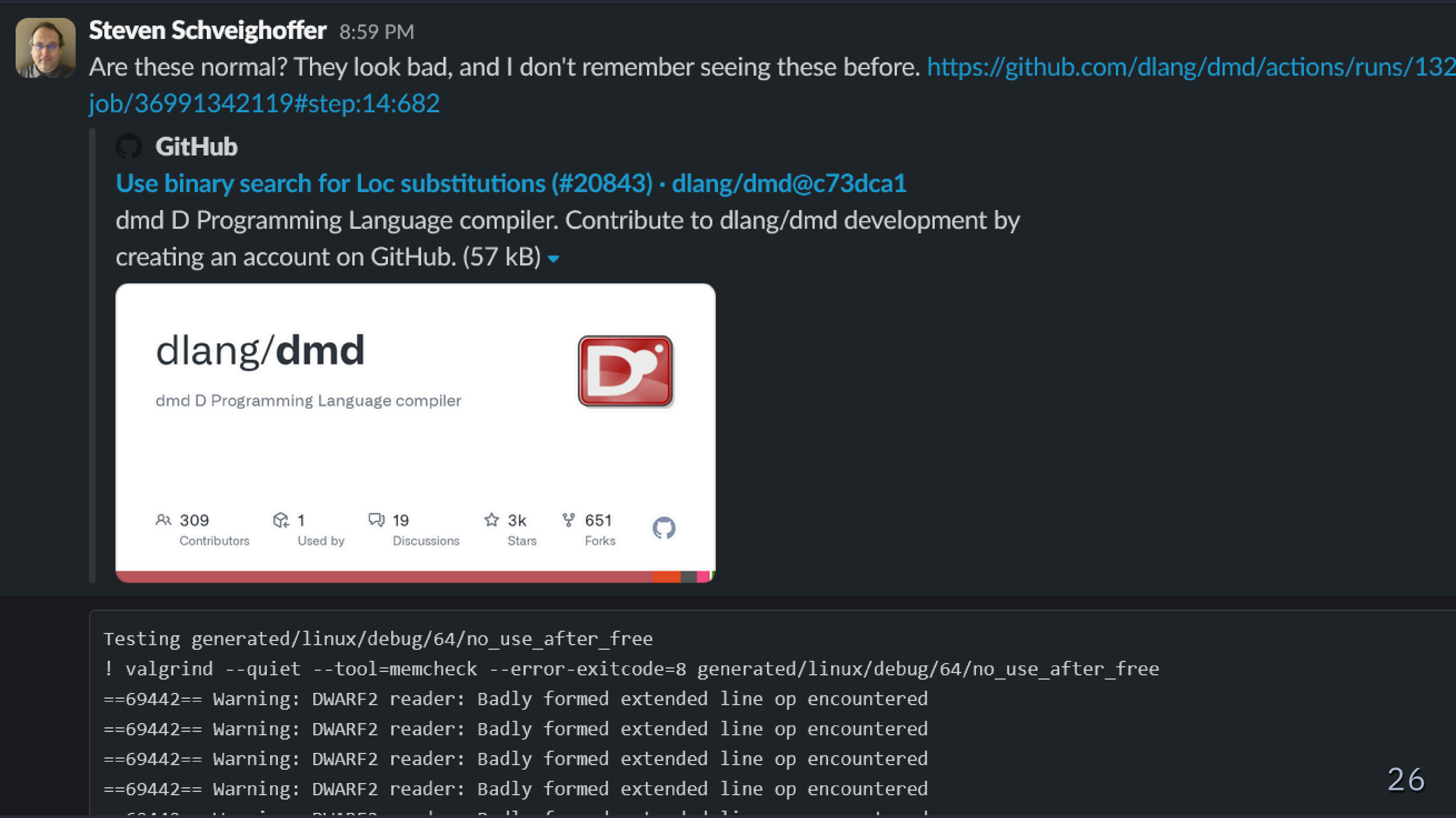
> Unreachable code is useless, it should be an error

- Have you considered: templates, conditional compilation, debugging, version control, dustmite...
- Yes, footguns like `if (x = 3)` exist
- But: removing composition = more complexity
  - Why is this combination useful? Let's ban it. ❌
  - GitHub/Forum links to bugs caused by this ✅

#3 - Avoid warnings

> “ If an error won't do, we could make it a warning instead ”

- Warnings pile up, get drowned out



Steven Schveighoffer 8:59 PM

Are these normal? They look bad, and I don't remember seeing these before. https://github.com/dlang/dmd/actions/runs/132
job/36991342119#step:14:682

GitHub

Use binary search for Loc substitutions (#20843) · dlang/dmd@c73dca1

dmd D Programming Language compiler. Contribute to dlang/dmd development by creating an account on GitHub. (57 kB) ▾

dlang/**dmd**

dmd D Programming Language compiler

309	1	19	3k	651
Contributors	Used by	Discussions	Stars	Forks

```
Testing generated/linux/debug/64/no_use_after_free
! valgrind --quiet --tool=memcheck --error-exitcode=8 generated/linux/debug/64/no_use_after_free
==69442== Warning: DWARF2 reader: Badly formed extended line op encountered
==69442== Warning: DWARF2 reader: Badly formed extended line op encountered
==69442== Warning: DWARF2 reader: Badly formed extended line op encountered
==69442== Warning: DWARF2 reader: Badly formed extended line op encountered
```

#3 - Avoid warnings

> Then treat warnings as errors in 'production builds'

- Has its own problems with false positives, updates, etc.

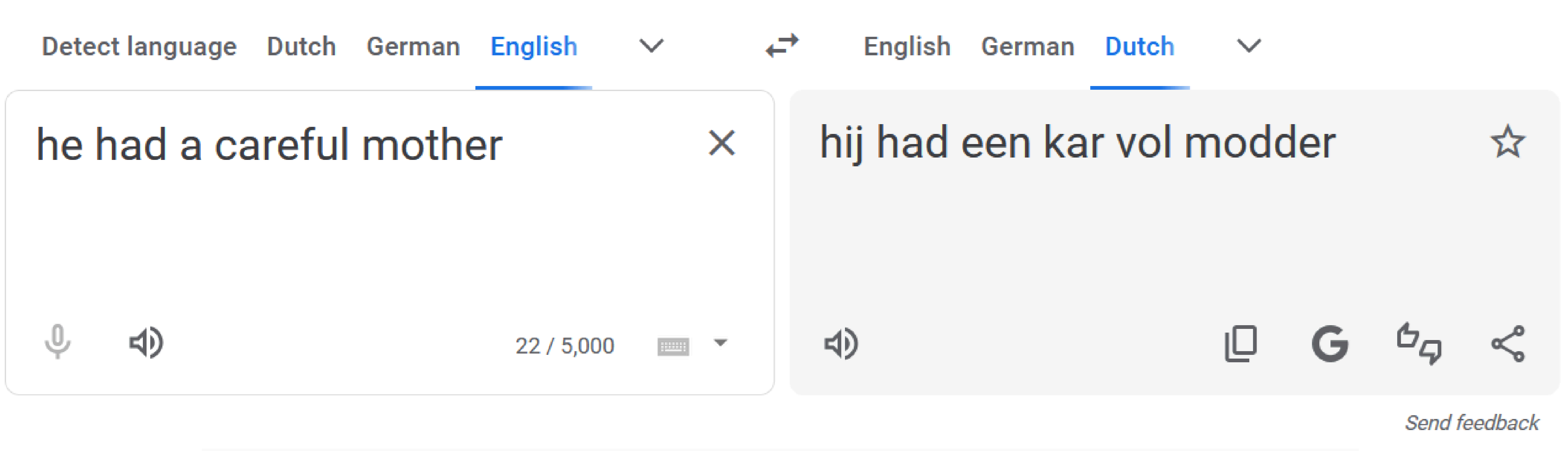
```
switch (x)
{
  ...
    case 1:
        abort();
        break; // ⚠️ unreachable code
  ...
}
```

#4 - More options ≠ better

> “ Can't be bad to give users the option with `-fno-unreachable-code` ”

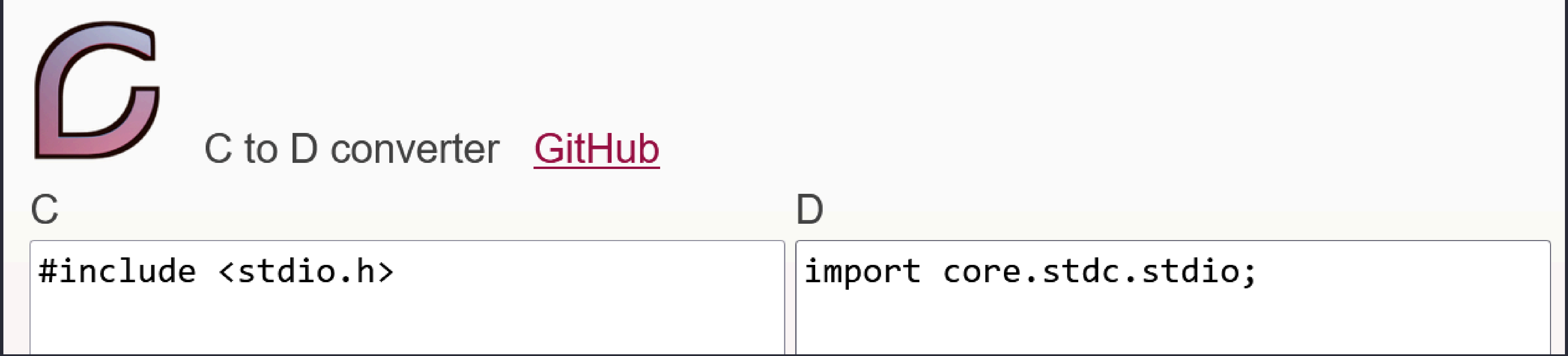
- Hot take: All command line switches are bugs
- Google Translate has 1 billion users
- So it must have *tons* of options?

```
google-translate
  --fix-spelling
  --word-wrap-columns=80
  --oxford-comma
  --custom-substitutions="onigiri/jelly-donut"
```



he had a careful mother

hij had een kar vol modder



```
#include <stdio.h>
```

```
import core.stdc.stdio;
```

https://translate.google.com/ · https://dkorpel.github.io/ctod/

#4 - More options ≠ better

- ctod used to have a `--strip-comments`

> Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".

- Now it has 0 flags
- Viable for dmd?

https://archive.org/details/bstj57-6-1899/mode/2up

#5 - Look for the root problem

- Often library solutions exist
- Disliked because:
  - Requires imports
  - Worse performance
  - Bad errors messages
  - Ugly syntax

#5 - Look for the root problem

- Often library solutions exist
- Disliked because:
  - Requires imports (prelude modules?)
  - Worse performance (optimized debug builds?)
  - Bad errors messages (diagnostic message attributes?)
  - Ugly syntax (new operator overloading?)

#5 - Look for the root problem

- Historical trends:
  - FORTRAN/COBOL → C/C++
  - Fixed graphics pipelines → shaders → GPGPU
  - Complex number type in C/D → SIMD, operator overloading
- Look for general building blocks

Better improvement proposals (conclusion)

- Motivate by real world problems
- Find the root cause
- Offer a confident solution
  - warnings/options should be last resort

Reducing technical debt

Picture: Lane7 Camden, London - "Play Dirty"

An unstable foundation supplies unlimited bug reports

- Whack-a-mole bug fixing
- `if (the_code == code_from_issue) do_the_desired_thing_instead()`
- Result: incpomplete, redundant solutions:
    - `ctfe`, `ctfeBlock`, `ctfeOnly`
    - `maybeScope`, `doNotInferScope`
- "The existing code was a hack, so I had to add my own hack"

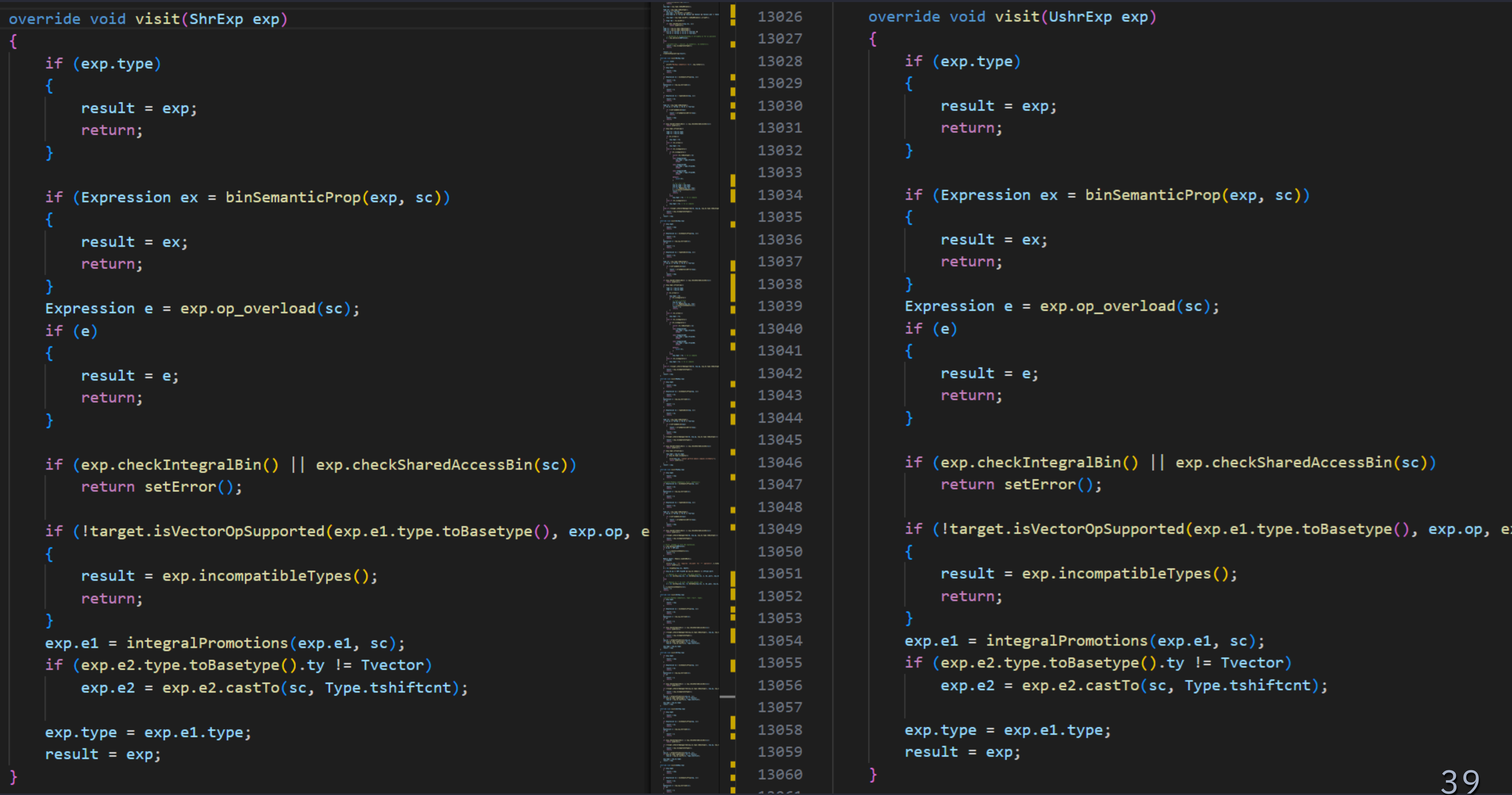
More passing test cases != progress

- Local optimum where common cases succeed
- Can be useful for experimentation
- At some point, sound solution must be found
- Wrong fixes must be undone

Example of undoing a wrong fix: https://github.com/dlang/dmd/pull/13972

Factor out common code

- Arithmetic operators type check almost identically
- Differences are often bugs
- Expression semantic for `>>>` and `>>` used to be copy-pasta



```
override void visit(ShrExp exp)
{
    if (exp.type)
    {
        result = exp;
        return;
    }

    if (Expression ex = binSemanticProp(exp, sc))
    {
        result = ex;
        return;
    }
    Expression e = exp.op_overload(sc);
    if (e)
    {
        result = e;
        return;
    }

    if (exp.checkIntegralBin() || exp.checkSharedAccessBin(sc))
        return setError();

    if (!target.isVectorOpSupported(exp.e1.type.toBasetype(), exp.op, e
    {
        result = exp.incompatibleTypes();
        return;
    }
    exp.e1 = integralPromotions(exp.e1, sc);
    if (exp.e2.type.toBasetype().ty != Tvector)
        exp.e2 = exp.e2.castTo(sc, Type.tshiftcnt);

    exp.type = exp.e1.type;
    result = exp;
}
```

```
override void visit(UshrExp exp)
{
    if (exp.type)
    {
        result = exp;
        return;
    }

    if (Expression ex = binSemanticProp(exp, sc))
    {
        result = ex;
        return;
    }
    Expression e = exp.op_overload(sc);
    if (e)
    {
        result = e;
        return;
    }

    if (exp.checkIntegralBin() || exp.checkSharedAccessBin(sc))
        return setError();

    if (!target.isVectorOpSupported(exp.e1.type.toBasetype(), exp.op, e
    {
        result = exp.incompatibleTypes();
        return;
    }
    exp.e1 = integralPromotions(exp.e1, sc);
    if (exp.e2.type.toBasetype().ty != Tvector)
        exp.e2 = exp.e2.castTo(sc, Type.tshiftcnt);

    exp.type = exp.e1.type;
    result = exp;
}
```

Reuse isn't always correct

- Consider `bool hasPointers(Type t)`
  - `int` → false
  - `int*` → true
  - `struct S { int x; string y; }` → true
  - `void[8]` → ?
- Depends! Conservative GC scanning or `@safe` checks?

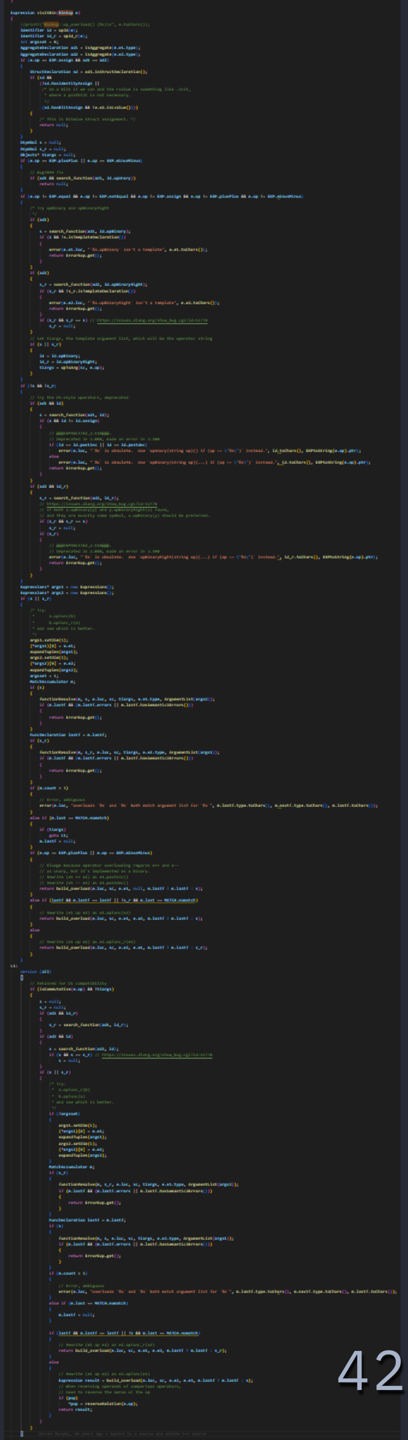
Avoid boolean parameters

```
bool hasPointers(Type t, bool usedForGcScanning)
{
    ...
    if (usedForGcScanning)
        if (t.kind == Tarray && t.next.kind == Tvoid)
            return true;
    ...
}
```

Spaghetti ensues

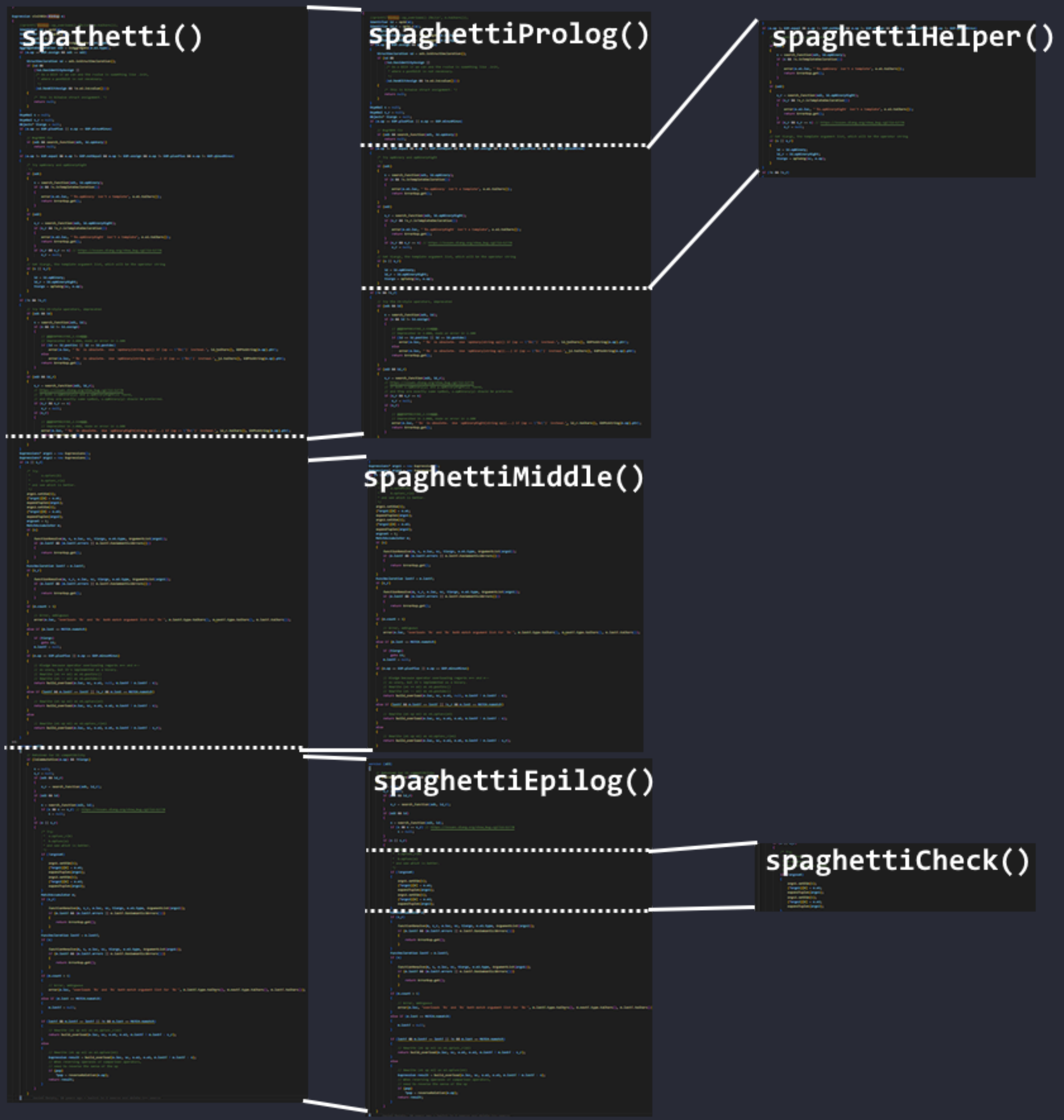
Semantic for `opAssign`, `opEquals`, `opBinary`, `opUnary`
All funneled through 300 line `overload()` function

```
if (e.op == EXP.plusPlus || e.op == EXP.minusMinus)
{
    // Bug4099 fix
    if (ad1 && search_function(ad1, Id.opUnary))
        return null;
}
if (e.op != EXP.equal && e.op != EXP.notEqual &&
    e.op != EXP.assign && e.op != EXP.plusPlus && e.op != EXP.minusMinus)
{
    // Try opBinary and opBinaryRight
}
```



Cutting up doesn't help

- Now you just have 5 incomprehensible functions
- Separate the code paths instead



```
Expression overload(Expression e)
{
    string name = "opBinary";
    if (e.op == "==")
        name = "opEquals";

    auto result = new CallExpression(name);
    if (e.op != "==")
        result.addTemplateArgs([e.op]);

    result.addArgs([e.lhs, e.rhs]);
    return result;
}
```

```
Expression overloadBinary(Expression e)
{
    string name = "opBinary";
    if (e.op == "==")
        name = "opEquals";

    auto result = new CallExpression(name);
    if (e.op != "==")
        result.addTemplateArgs([e.op]);

    result.addArgs([e.lhs, e.rhs]);
    return result;
}
```

```
Expression overloadBinary(Expression e)
{
    string name = "opBinary";
    if (false)
        name = "opEquals";

    auto result = new CallExpression(name);
    if (true)
        result.addTemplateArgs([e.op]);

    result.addArgs([e.lhs, e.rhs]);
    return result;
}
```

```
Expression overloadBinary(Expression e)
{
    string name = "opBinary";
    auto result = new CallExpression(name);
    if (true)
        result.addTemplateArgs([e.op]);

    result.addArgs([e.lhs, e.rhs]);
    return result;
}
```

```
Expression overloadBinary(Expression e)
{
    string name = "opBinary";
    auto result = new CallExpression(name);

        result.addTemplateArgs([e.op]);

    result.addArgs([e.lhs, e.rhs]);
    return result;
}
```

```
Expression overloadBinary(Expression e)
{
    string name = "opBinary";
    auto result = new CallExpression(name);
    result.addTemplateArgs([e.op]);
    result.addArgs([e.lhs, e.rhs]);
    return result;
}
```

```
Expression overloadBinary(Expression e)
{
    string name = "opBinary";
    auto result = new CallExpression(name);
    result.addTemplateArgs([e.op]);
    result.addArgs([e.lhs, e.rhs]);
    return result;
}
```

```
Expression overloadBinary(Expression e)
{
    auto result = new CallExpression("opBinary");
    result.addTemplateArgs([e.op]);
    result.addArgs([e.lhs, e.rhs]);
    return result;
}
```

```
Expression overloadEquals(Expression e)
{
    auto result = new CallExpression("opEquals");
    result.addArgs([e.lhs, e.rhs]);
    return result;
}
```

```d
Expression overloadBinary(Expression e)
{
    return callOpOverload("opBinary", [e.op], [e.lhs, e.rhs]);
}

Expression overloadEquals(Expression e)
{
    return callOpOverload("opEquals", [], [e.lhs, e.rhs]);
}

Expression callOpOverload(string name, Expression[] tiArgs, Expression[] args)
{
    auto result = new CallExpression(name);
    result.addTemplateArgs(tiArgs);
    result.addArg(args);
    return result;
}
```

```
Expression overload(Expression e)
{
    string name = "opBinary";
    if (e.op == "==")
        name = "opEquals";

    auto result = new CallExpression(name);
    if (e.op != "==")
        result.addTemplateArgs([e.op]);

    result.addArg(e.lhs);
    result.addArg(e.rhs);
    return result;
}
```

Reducing technical debt (conclusion)

- Code duplication and premature abstraction can both be bad
- When you can't get away with duct tape solutions:
  - **Expand** intertwined code paths
  - **Trim** dead branches
  - **Factor out** common code again

https://dlang.org/changelog/2.111.0.html#dmd.error-messages

Takeaways

- There is a limited complexity budget for features
- Strong proposals spend little to solve real problems
- Pay off technical debt to expand your budget
- Don't take the "No" personal

Questions?