Tour of the DMD D Programming Language Compiler

by Walter Bright

http://www.walterbright.com


IBM

Organization

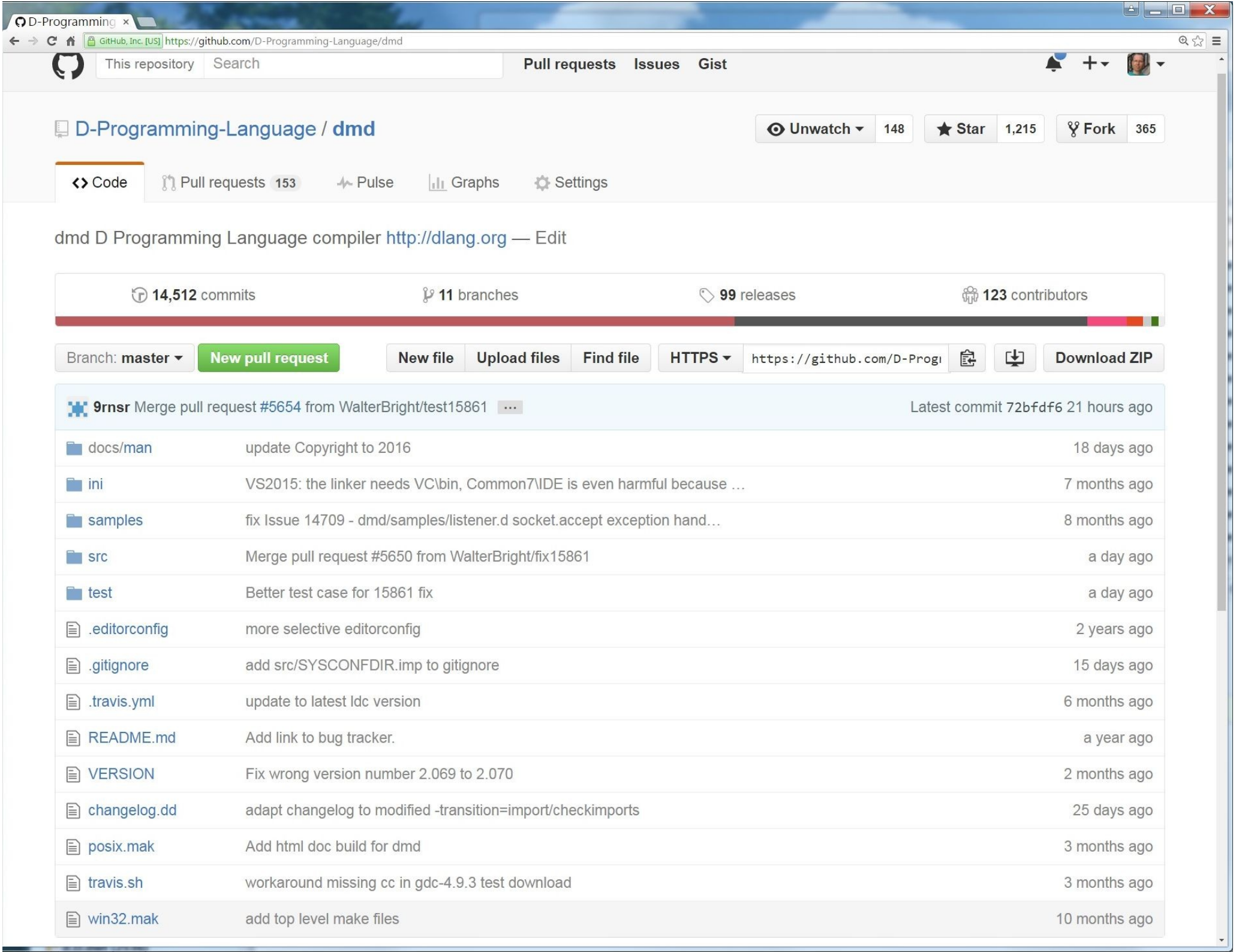
- One front end to rule them all
  - and in the D bind them
  - written in D
- Three back ends
  - Digital Mars (dmd)
  - Gnu compiler collection (gdc)
  - LLVM (ldc)

Major Changes in Last Year

- Converted front end from C++ to D
- Switch to using Dwarf exception handling
  - opens up more comprehensive C++ interfacing

Source Code

https://github.com/dlang/dmd


D-Programming
GitHub, Inc. [US] https://github.com/D-Programming-Language/dmd
This repository
Search
Pull requests
Issues
Gist
D-Programming-Language / dmd
Unwatch 148
Star 1,215
Fork 365
Code
Pull requests 153
Pulse
Graphs
Settings
dmd D Programming Language compiler http://dlang.org — Edit
14,512 commits
11 branches
99 releases
123 contributors
Branch: master
New pull request
New file
Upload files
Find file
HTTPS
https://github.com/D-Progı
Download ZIP
9rnsr Merge pull request #5654 from WalterBright/test15861 ···
Latest commit 72bfdf6 21 hours ago
docs/man update Copyright to 2016 18 days ago
ini VS2015: the linker needs VC\bin, Common7\IDE is even harmful because … 7 months ago
samples fix Issue 14709 - dmd/samples/listener.d socket.accept exception hand… 8 months ago
src Merge pull request #5650 from WalterBright/fix15861 a day ago
test Better test case for 15861 fix a day ago
.editorconfig more selective editorconfig 2 years ago
.gitignore add src/SYSCONFDIR.imp to gitignore 15 days ago
.travis.yml update to latest ldc version 6 months ago
README.md Add link to bug tracker. a year ago
VERSION Fix wrong version number 2.069 to 2.070 2 months ago
changelog.dd adapt changelog to modified -transition=import/checkimports 25 days ago
posix.mak Add html doc build for dmd 3 months ago
travis.sh workaround missing cc in gdc-4.9.3 test download 3 months ago
win32.mak add top level make files 10 months ago

Directories

- `src/` front end source code
- `src/tk` generic code for back end
- `src/root` generic code for front end
- `src/backend` optimizer and code generator
- `src/vcbuild` for building compiler with VS

Types of Compiles

- diagnose errors in source code
- generate a debug build
- generate a release build

Memory Allocation

- `root/rmem.d`
- Allocate, but never free
- Very fast
- No need for ownership tracking
- Puts upper limit on size of compile

Strings

- `root/stringtable.d`
- `const(char)*`
- identifier strings stored in single hash table
- address of string becomes its hash
  - `Identifier.idPool()`
- very fast

Array(T)

- root/array.d
- a workalike to D dynamic arrays
- accessible from C++ code
- heavily used

```
alias Strings = Array!(const(char)*);
alias Statements = Array!Statement;
alias Identifiers = Array!Identifier;
… etc.
```

RootObject

- `root/object.d`
- single rooted hierarchy
  - much like D's `Object`, but predates it
  - a C++ class, so accessible from glue layer
  - `Declarations`, `Statements`, `Expressions`
  - heavy use of OOP plus Visitor Pattern

Passes

- read files
- lex
- parse
- create symbol table
- semantic 1
- semantic 2
- semantic 3
- inline
- glue
- optimize
- generate code
- write to object file

Lexing

- `lexer.d`
- pretty simple
- rarely changes
- mostly concerned with speed

Parsing

- parse.d
- also simple and rarely changes
- lookahead is done by forming a stack of tokens
- code looks a lot like the grammar …

```
case TOKwhile:
 {
   nextToken();
   check(TOKlparen);
   Expression condition = parseExpression();
   check(TOKrparen);
   Loc endloc;
   Statement _body =
       parseStatement(PSscope, null, &endloc);
   s = new
       WhileStatement(loc,condition,_body,endloc);
   break;
 }
```

Create Symbol Table

- `importAll()`
- establishes a `Scope` for each symbol

`Scope`

- `dscope.d`
- link to enclosing `Scope`
- fields
  - module
  - function
  - storage class in effect
  - ...

Semantic

```
int a;
int b = 3;
int foo() {
      return 6;
}
```

Lowering

- rewriting ASTs to simpler, canonical forms
- reduces number of cases needing to be dealt with later
- reduces complexity and bugs
- even makes it easier to document

Loops

```
while (cond) { body }

for (; cond; ) { body }

foreach (i; n .. m) { body }

for (auto i = n; i < m; ++i) { body }

foreach (e; aggr) { body }

for (auto r = aggr[]; !r.empty; r.popFront())
{
  auto e = r.front;
  body;
}
```

Exceptions

- rewritten to be try-finally
- scope
- synchronized
- RAII

Error Recovery Models

- Quit on first error
- Guess at user intention, then repair
- Poisoning

Poisoning

- have a special 'error' AST node
- replace erroneous AST node with 'error' node
- replace any node that has an 'error' child with an 'error' node
- virtually eliminates cascaded errors
  - errors displayed are real errors

Spell Checking

- `root/speller.d`
- for undefined identifiers

Constant Folding

- constfold.d

```
UnionExp Bool(Type type, Expression e1) {
  UnionExp ue;
  Loc loc = e1.loc;
  emplaceExp!(IntegerExp)(&ue, loc, e1.isBool(true)?1:0, type);
  return ue;
}
```

Compile Time Function Execution (CTFE)

- just a glorified constant folder
- allocates memory to evaluate an expression
- so it runs out of memory
- and is slow

Templates

- Stored as ASTs as produced by the parser
- To instantiate:
  - copy the AST
  - set the scope to where the template declaration is in the symbol table
  - create symbols from template arguments
  - run semantic() passes

Inlining

- inline.d
- functions that can be represented as an expression can be inlined

```
int func(int x) { if (x == 8) return 9; else return 68; }

y = func(z) + 8;

y = ((int x = z),(x == 8 ? 9 : 68)) + 8;
```

- but that doesn't work with loops

Inlining Statements

```
x = 3;
func(x);
y = x + 3;
```

Challenges

- eliminate global variables
  - sooner or later **always** cause trouble with recursive logic like compiler guts
- get a grip on complexity
  - reduce cyclomatic complexity
  - code should *flow* rather than hop around
  - change data structures to eliminate special cases
- reduce memory consumption
  - localize (i.e. encapsulate) memory management

More Challenges

- improve encapsulation
  - containers leak implementation details like being a linked list or an array
  - encapsulation means data structures can be changed
- use `const` / `pure` / `nothrow` / `@safe`
- better dogfooding
  - too many vestiges of the older C++ implementation hanging around

Conclusion

- I like working on compilers
- It never stops being fun (much more fun than playing video games!)
- Always learning new ways to make the code better
- All welcome to fork on Github and join in the fray!