# Getting up, running, and contributing to GHC

**Ryan Scott**

Galois, Inc.

GHC Contributors' Workshop
7 June, 2023

# About me

- Learned Haskell in 2013, now use it professionally
- First GHC contribution in 2015
- Expertise is in GHC’s frontend (`deriving`, Template Haskell, pattern matching, typechecking, etc.)

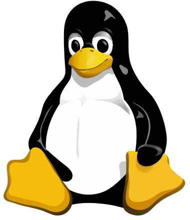
# Why contribute to GHC?

- You *can* make a difference
- Improve your understanding on the language and tools
- Your contributions help everyone (including yourself)
- It’s fun!

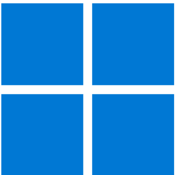
# Preparing to build GHC

**https://gitlab.haskell.org/ghc/ghc/-/wikis/building/preparation**

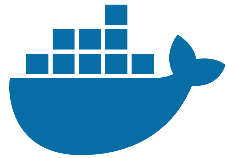
# Supported configurations


Linux

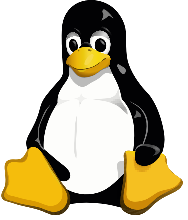
macOS


Windows

Nix

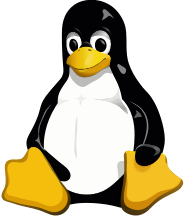

Docker

# Supported configurations



Linux

# Supported configurations



Linux

```
$ sudo apt-get install build-essential
    git autoconf python3 libgmp-dev
    libnuma-dev libncurses-dev

$ cabal v2-install alex happy

$ ghcup install ghc 9.4.5
```

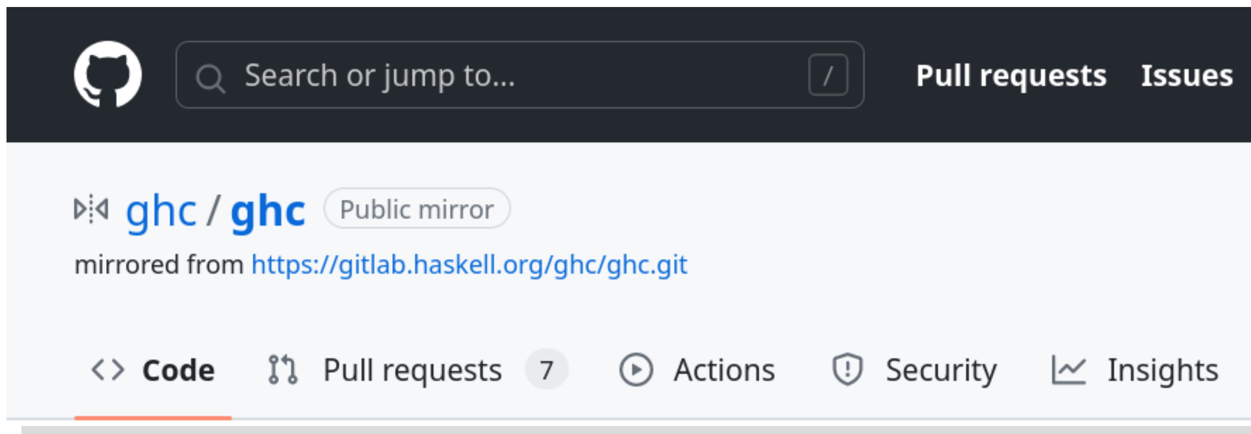
# Cloning GHC

# Cloning GHC

```
$ git clone --recurse-submodules
    https://gitlab.haskell.org/ghc/ghc.git
```

# Cloning GHC

```
$ git clone --recurse-submodules
    https://gitlab.haskell.org/ghc/ghc.git
```



# Managing multiple GHC trees

# Managing multiple GHC trees

- Option 1: Have separate checkouts for each GHC feature you develop

# Managing multiple GHC trees

- Option 1: Have separate checkouts for each GHC feature you develop

- Option 2: Use `git wtas` to manage multiple working trees within the same checkout:

```
$ git wtas ../ghc-my-new-feature
$ git submodule update --init
```

(`git wtas` is defined at `https://stackoverflow.com/a/31872051/388010`)

# Build system

# Hadrian

- Custom-made build system based on Shake library

- https://gitlab.haskell.org/ghc/ghc/blob/master/hadrian/README.md

# Hadrian: Your first build

# Hadrian: Your first build

```
$ ./boot && ./configure      # run autoconf scripts, etc.
```

# Hadrian: Your first build

```
$ ./boot && ./configure --enable-tarballs-autodownload
  # Windows-only flag for downloading external dependencies
```

# Hadrian: Your first build

```
$ ./boot && ./configure      # run autoconf scripts, etc.
```

# Hadrian: Your first build

```
$ ./boot && ./configure       # run autoconf scripts, etc.

$ hadrian/build -j            # build GHC with parallelism
```

# Hadrian: Your first build

```
$ ./boot && ./configure      # run autoconf scripts, etc.

$ hadrian/build -j           # build GHC with parallelism

# Go brew some coffee and wait :)
```

# Hadrian: Your first build

```
$ ./boot && ./configure        # run autoconf scripts, etc.

$ hadrian/build -j             # build GHC with parallelism

# Go brew some coffee and wait :)

$ _build/stage1/bin/ghc --version
The Glorious Glasgow Haskell Compilation System, version
9.7.20230430
```

# Hadrian: Using GHCi

# Hadrian: Using GHCi

# Hadrian: Using GHCi

```
$ _build/stage1/bin/ghci
```

```
$ ls _build/stage1/bin
ghc   ghc-pkg   haddock   hp2ps   hpc   hsc2hs   runghc
```

# Hadrian: Using GHCi

```
$ _build/stage1/bin/ghci
```

```
$ ls _build/stage1/bin
ghc   ghc-pkg   haddock   hp2ps   hpc   hsc2hs   runghc

$ _build/stage1/bin/ghc --interactive
```

# Hadrian: Build stages

| Stage 0 | Stage 1 | Stage 2 | Stage 3 |
| --- | --- | --- | --- |
| | | | |

# Hadrian: Build stages

| Stage 0 | Stage 1 | Stage 2 | Stage 3 |
| --- | --- | --- | --- |

# Hadrian: Build stages

Stage 0 Stage 1 Stage 2 Stage 3

| Stage 0 | Stage 1 | Stage 2 | Stage 3 |
|---|---|---|---|
| Installed GHC (bootstrap compiler) | | | |

# Hadrian: Build stages

| Stage 0 | Stage 1 | Stage 2 | Stage 3 |
|---|---|---|---|
| Installed GHC (bootstrap compiler) | Basic GHC with core libraries | | |

# Hadrian: Build stages

| Stage 0 | Stage 1 | Stage 2 | Stage 3 |
| --- | --- | --- | --- |
| Installed GHC (bootstrap compiler) | Basic GHC with core libraries | Complete GHC with all libraries installed | |

# Hadrian: Build stages

| Stage 0 | Stage 1 | Stage 2 | Stage 3 |
| --- | --- | --- | --- |
| Installed GHC (bootstrap compiler) | Basic GHC with core libraries | Complete GHC with all libraries installed | (Optional) Useful for cross-compiling |

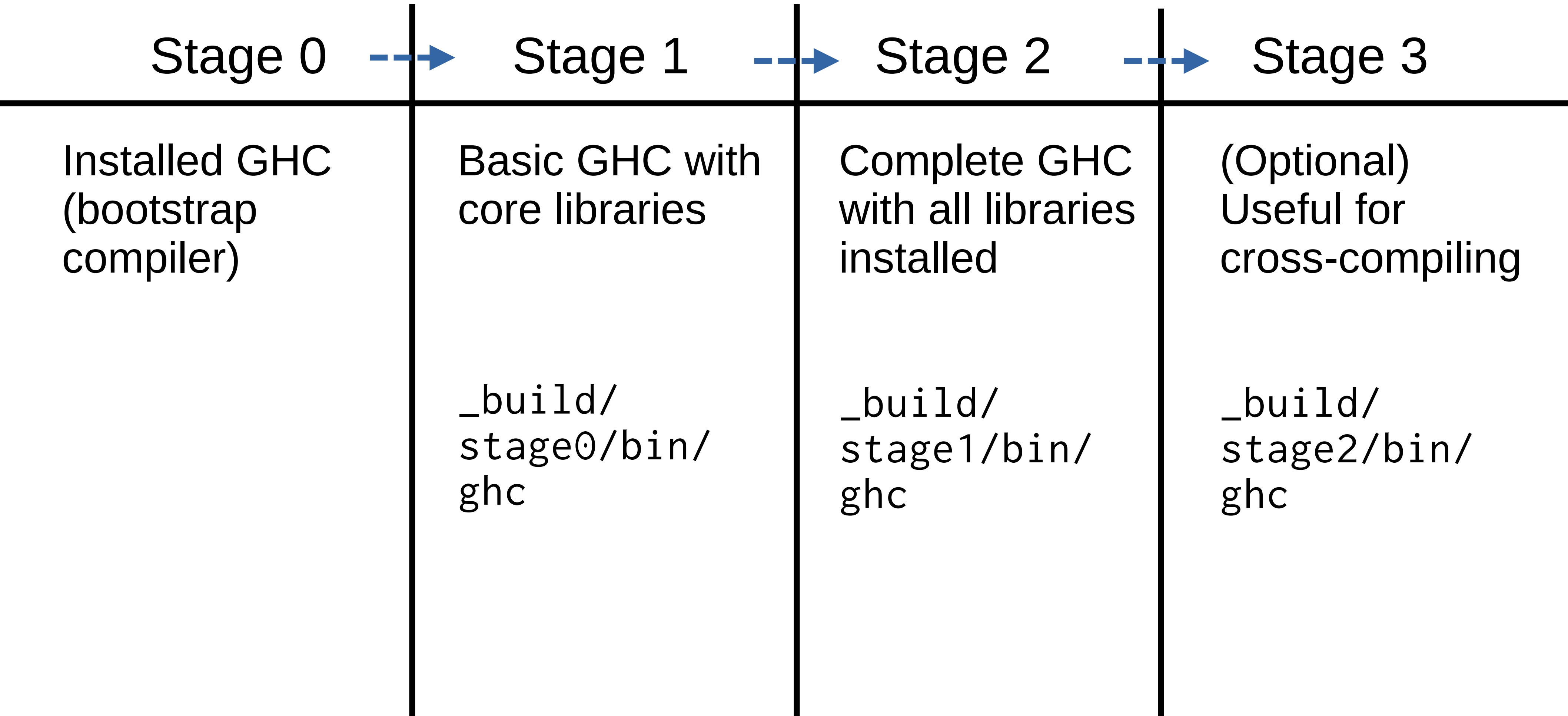
# Hadrian: Build stages

| Stage 0 | Stage 1 | Stage 2 | Stage 3 |
| --- | --- | --- | --- |
| Installed GHC (bootstrap compiler) | Basic GHC with core libraries<br><br>`_build/stage0/bin/ghc` | Complete GHC with all libraries installed | (Optional) Useful for cross-compiling |

# Hadrian: Build stages

| Stage 0 | Stage 1 | Stage 2 | Stage 3 |
| --- | --- | --- | --- |
| Installed GHC (bootstrap compiler) | Basic GHC with core libraries<br><br>`_build/stage0/bin/ghc` | Complete GHC with all libraries installed | (Optional) Useful for cross-compiling |

# Hadrian: Build stages



# Hadrian: Build flavours

`--flavour=FLAVOUR`: Configures the build settings

- Default is `--flavour=default`
- Others include `quick`, `perf`, `prof`, `devel1`, `devel2`, etc.

# Hadrian: Build flavours

`--flavour=FLAVOUR`: Configures the build settings

- Default is `--flavour=default`
- Others include `quick`, `perf`, `prof`, `devel1`, `devel2`, etc.

Also *flavour transformers* for modifying existing flavours

- Examples include `werror`, `debug_info`, etc.
- e.g., `--flavour=default+werror`

# Hadrian: Build flavours

`--flavour=FLAVOUR`: Configures the build settings

- Default is `--flavour=default`
- Others include `quick`, `perf`, `prof`, `devel1`, `devel2`, etc.

Also *flavour transformers* for modifying existing flavours

- Examples include `werror`, `debug_info`, etc.
- e.g., `--flavour=default+werror`

**The “u” in “flavour” is mandatory!**

# Hadrian: Build flavours

# Hadrian: Build flavours

```
$ hadrian/build --flavour=quick

$ hadrian/build
```

# Hadrian: Build flavours

```
$ hadrian/build --flavour=quick

$ hadrian/build
```

# Hadrian: Build flavours

```
$ hadrian/build --flavour=quick

$ hadrian/build --flavour=quick
```

# Hadrian: User settings

- Specify flavour to use on every Hadrian invocation by copying the file:

  `hadrian/src/UserSettings.hs`

  To:

  `hadrian/UserSettings.hs`

# Hadrian: Save yourself some time

`--freeze1`: Once stage-1 GHC is built, do not rebuild it on subsequent invocation of Hadrian

- Significantly reduces rebuild times

# Hadrian: Save yourself some time

`--freeze1`: Once stage-1 GHC is built, do not rebuild it on subsequent invocation of Hadrian

- Significantly reduces rebuild times

`hadrian/ghci`: Load GHC's source code into GHCi session

- Useful for fast development feedback
- Note that code is only typechecked, not compiled, so you cannot run GHC itself this way

# Hadrian: Running the test suite

```
$ hadrian/build test
```

# Hadrian: Running the test suite

```
$ hadrian/build test

$ hadrian/build test --only="test1 test2"
```

# Hadrian: Haskell Language Server (HLS)

- GHC’s source code generally Just Works™ with HLS
- Loading the GHC source code into HLS for the first time can take a while, so you can use this to “pre-build” it:

```
$ hadrian/build --build-root=.hie-bios --flavour=ghc-in-ghci
    --docs=none -j tool:ghc/Main.hs
```

# Code overview

```
$ tree ghc -L 1 -d
ghc
├── compiler
├── docs
├── ghc
├── hadrian
├── libraries
├── nofib
├── rts
├── testsuite
└── utils
```

```
$ tree ghc -L 1 -d
ghc
├── compiler
├── docs
├── ghc
├── hadrian
├── libraries
├── nofib
├── rts
├── testsuite
└── utils
```

Build system

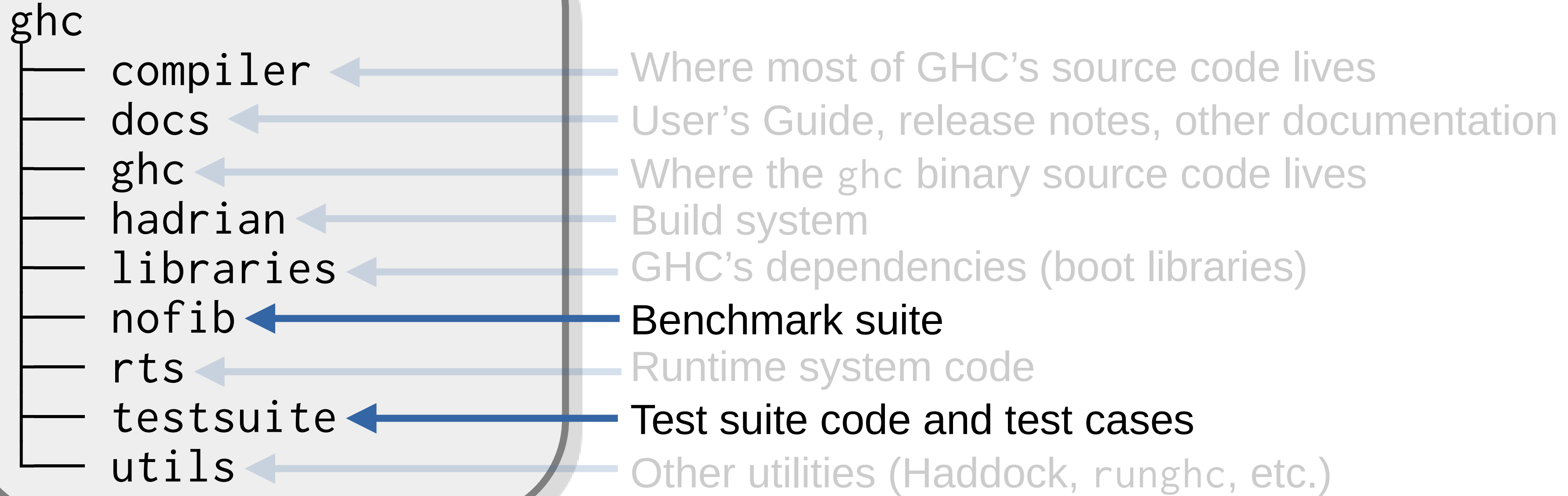
```
$ tree ghc -L 1 -d
ghc
├── compiler
├── docs
├── ghc
├── hadrian
├── libraries
├── nofib
├── rts
├── testsuite
└── utils
```

compiler: Where most of GHC’s source code lives

ghc: Where the `ghc` binary source code lives

hadrian: Build system

```
$ tree ghc -L 1 -d
ghc
├── compiler
├── docs
├── ghc
├── hadrian
├── libraries
├── nofib
├── rts
├── testsuite
└── utils
```


$ tree ghc -L 1 -d
ghc
compiler
docs
ghc
hadrian
libraries
nofib
rts
testsuite
utils
Where most of GHC's source code lives
User's Guide, release notes, other documentation
Where the ghc binary source code lives
Build system
GHC's dependencies (boot libraries)
Benchmark suite
Runtime system code
Test suite code and test cases
Other utilities (Haddock, runghc, etc.)

# Code style

# Capitalization conventions

# Capitalization conventions

```
tcLookupRecSelParent (RnRecUpdParent { rnRecUpdCons = cons })
  = case any_con of
      PatSynName ps ->
        RecSelPatSyn <$> tcLookupPatSyn ps
      DataConName dc ->
        RecSelData . dataConTyCon <$> tcLookupDataCon dc
  where
    any_con = head $ nonDetEltsUniqSet cons
```

# Capitalization conventions

```
tcLookupRecSelParent (RnRecUpdParent { rnRecUpdCons = cons })
  = case any_con of
      PatSynName ps ->
        RecSelPatSyn <$> tcLookupPatSyn ps
      DataConName dc ->
        RecSelData . dataConTyCon <$> tcLookupDataCon dc
  where
    any_con = head $ nonDetEltsUniqSet cons
```

# Capitalization conventions

```
tcLookupRecSelParent (RnRecUpdParent { rnRecUpdCons = cons })
  = case any_con of
      PatSynName ps ->
        RecSelPatSyn <$> tcLookupPatSyn ps
      DataConName dc ->
        RecSelData . dataConTyCon <$> tcLookupDataCon dc
  where
    any_con = head $ nonDetEltsUniqSet cons
```

CamelCase

- Exported identifiers

snake_case

- Non-exported identifiers
- Local identifiers (i.e., with `let` or `where`)

# Whitespace vs. semicolons

# Whitespace vs. semicolons

```
doptM flag = do
  logger <- getLogger
  return (logHasDumpFlag logger flag)
```

# Whitespace vs. semicolons

```haskell
doptM flag = do
  logger <- getLogger
  return (logHasDumpFlag logger flag)
```

```haskell
getEps = do
  { env <- getTopEnv
  ; liftIO $ hscEPS env
  }
```

# Documentation conventions

- Make sure to leave enough comments for others to understand the code you contribute
- GHC uses *Notes* for long-form comments

# Documentation conventions

- Make sure to leave enough comments for others to understand the code you contribute
- GHC uses *Notes* for long-form comments

```haskell
-- | Pretty-prints a 'TyThing'.
pprTyThing :: ShowSub -> TyThing -> SDoc
-- We pretty-print 'TyThing' via 'IfaceDecl'
-- See Note [Pretty printing via Iface syntax]
```

```haskell
pprIfaceDecl :: ShowSub -> IfaceDecl -> SDoc
-- NB: pprIfaceDecl is also used for pretty-printing TyThings in GHCi
--     See Note [Pretty printing via Iface syntax] in GHC.Types.TyThing.Ppr
```

# Documentation conventions

- Make sure to leave enough comments for others to understand the code you contribute
- GHC uses *Notes* for long-form comments

```
-- | Pretty-prints a 'TyThing'.
pprTyThing :: ShowSub -> TyThing -> SDoc
-- We pretty-print 'TyThing' via 'IfaceDecl'
-- See Note [Pretty printing via Iface syntax]
```

```
pprIfaceDecl :: ShowSub -> IfaceDecl -> SDoc
-- NB: pprIfaceDecl is also used for pretty-printing TyThings in GHCi
--     See Note [Pretty printing via Iface syntax] in GHC.Types.TyThing.Ppr
```

# Documentation conventions

- Make sure to leave enough comments for others to understand the code you contribute
- GHC uses *Notes* for long-form comments

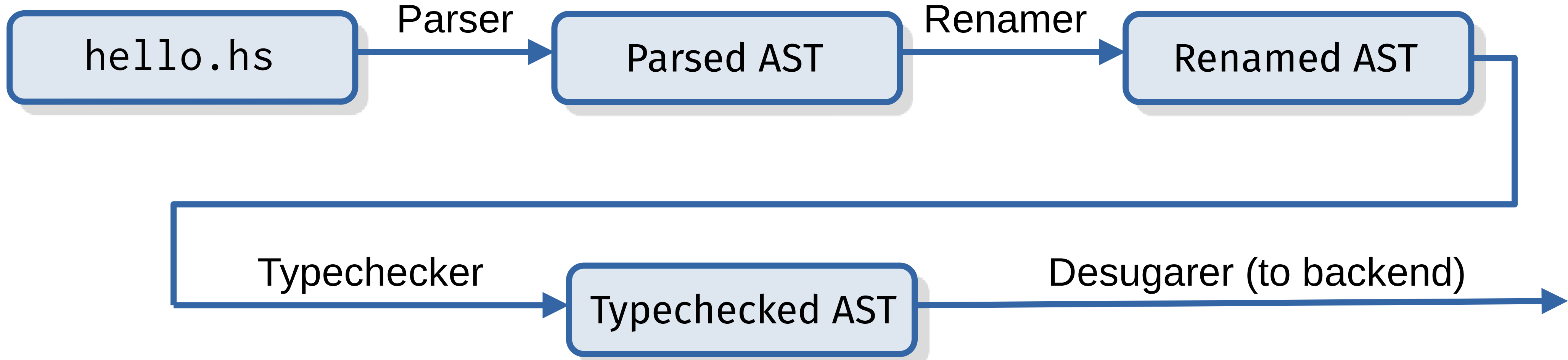
```
{- Note [Pretty printing via Iface syntax]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Our general plan for pretty-printing
  - Types
  - TyCons
  - Classes
  - Pattern synonyms
  ...etc...

is to convert them to Iface syntax, and pretty-print that. For example
  - pprType converts a Type to an IfaceType, and pretty prints that.
  - pprTyThing converts the TyThing to an IfaceDecl,
    and pretty prints that.
```

```
-- | Pretty-prints a 'TyThing'.
pprTyThing :: ShowSub -> TyThing -> SDoc
-- We pretty-print 'TyThing' via 'IfaceDecl'
-- See Note [Pretty printing via Iface syntax]
```

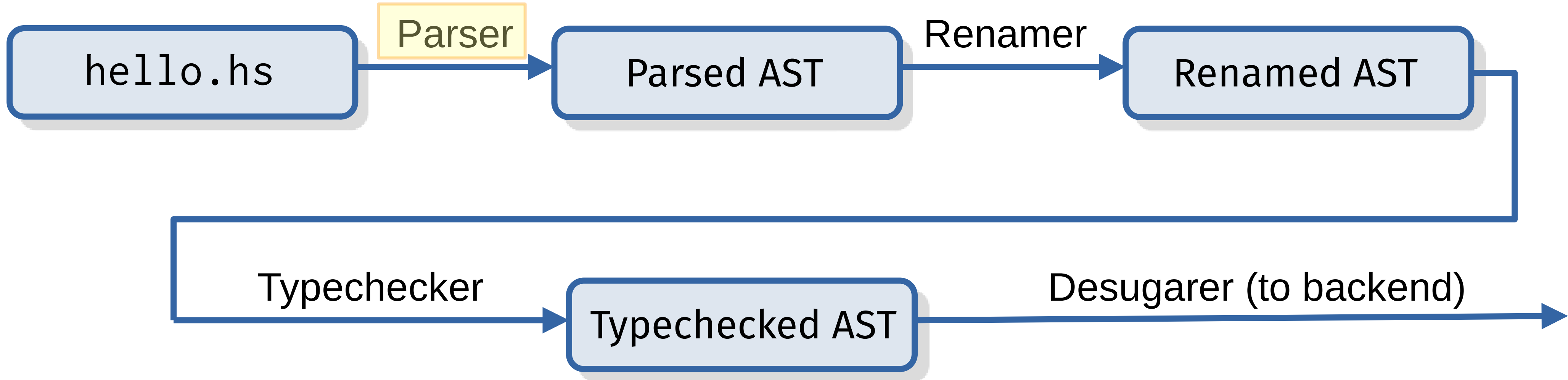
```
pprIfaceDecl :: ShowSub -> IfaceDecl -> SDoc
-- NB: pprIfaceDecl is also used for pretty-printing TyThings in GHCi
--     See Note [Pretty printing via Iface syntax] in GHC.Types.TyThing.Ppr
```

# Compiler pipeline

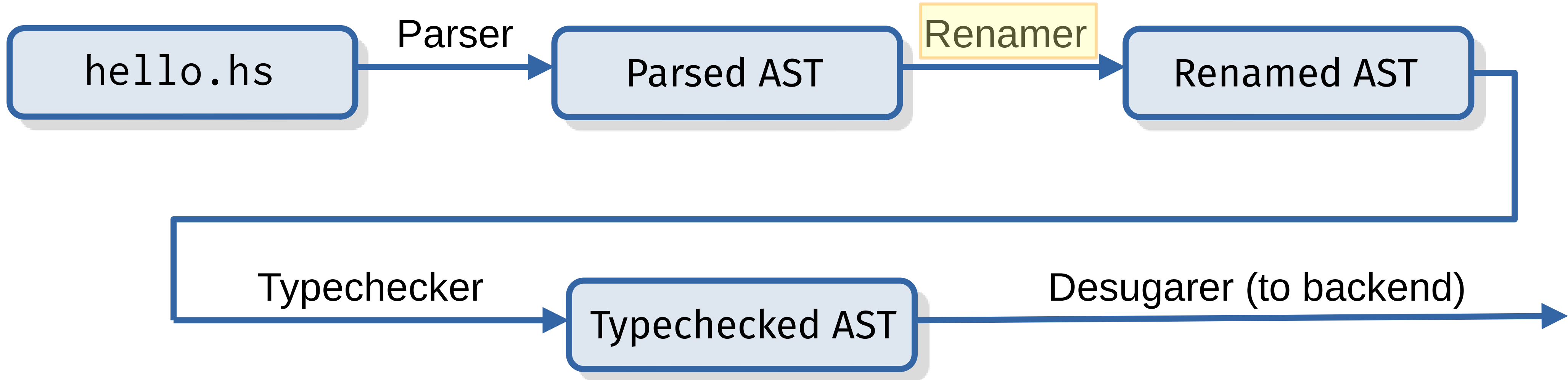
# Compiler pipeline (frontend)



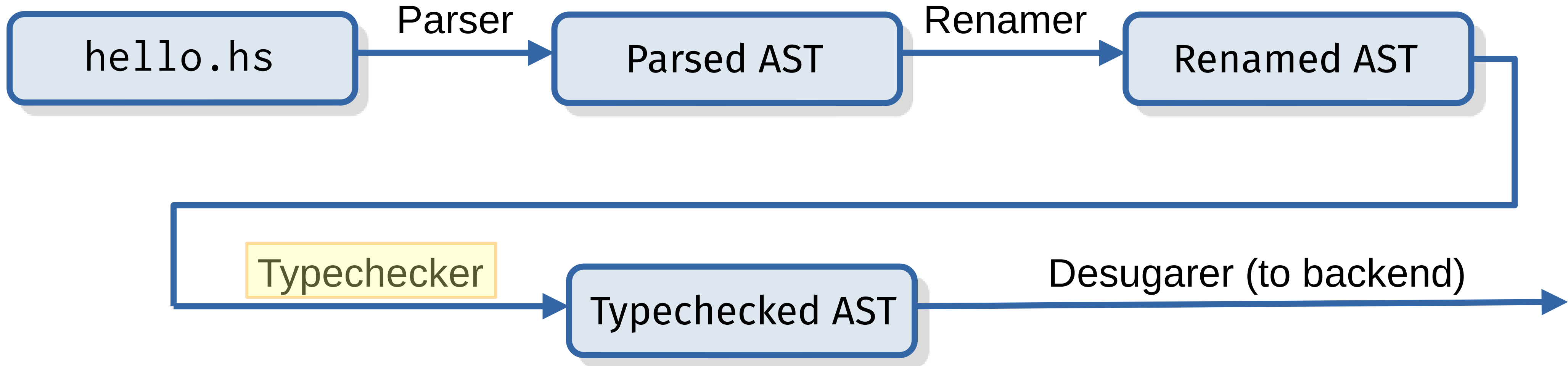
# Compiler pipeline (frontend)



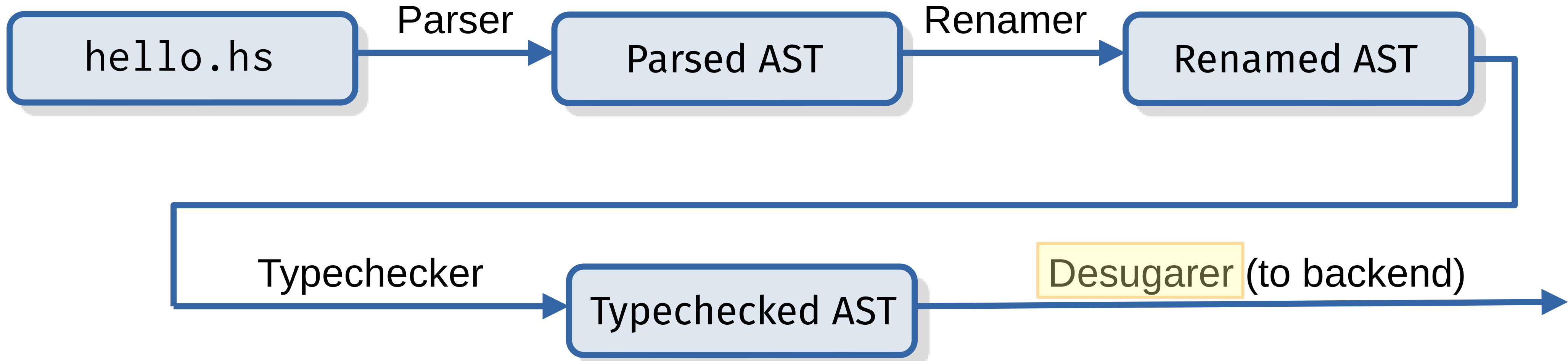
# Compiler pipeline (frontend)



# Compiler pipeline (frontend)



# Compiler pipeline (frontend)



# Compiler pipeline (frontend)

- Key data types: Haskell ASTs in `compiler/Language/Haskell/Syntax`:

# Compiler pipeline (frontend)

- Key data types: Haskell ASTs in `compiler/Language/Haskell/Syntax`:

```
data HsExpr p
  = HsVar (XVar p) (LIdP p)
  | HsApp (XApp p) (LHsExpr p) (LHsExpr p)
  | HsLam (XLam p)
          (MatchGroup p (LHsExpr p))
  | ...
```

```
data HsType p
  = HsTyVar (XTyVar p)
            PromotionFlag (LIdP p)
  | HsAppTy (XAppTy p)
            (LHsType p) (LHsType p)
  | ...
```

# Compiler pipeline (frontend)

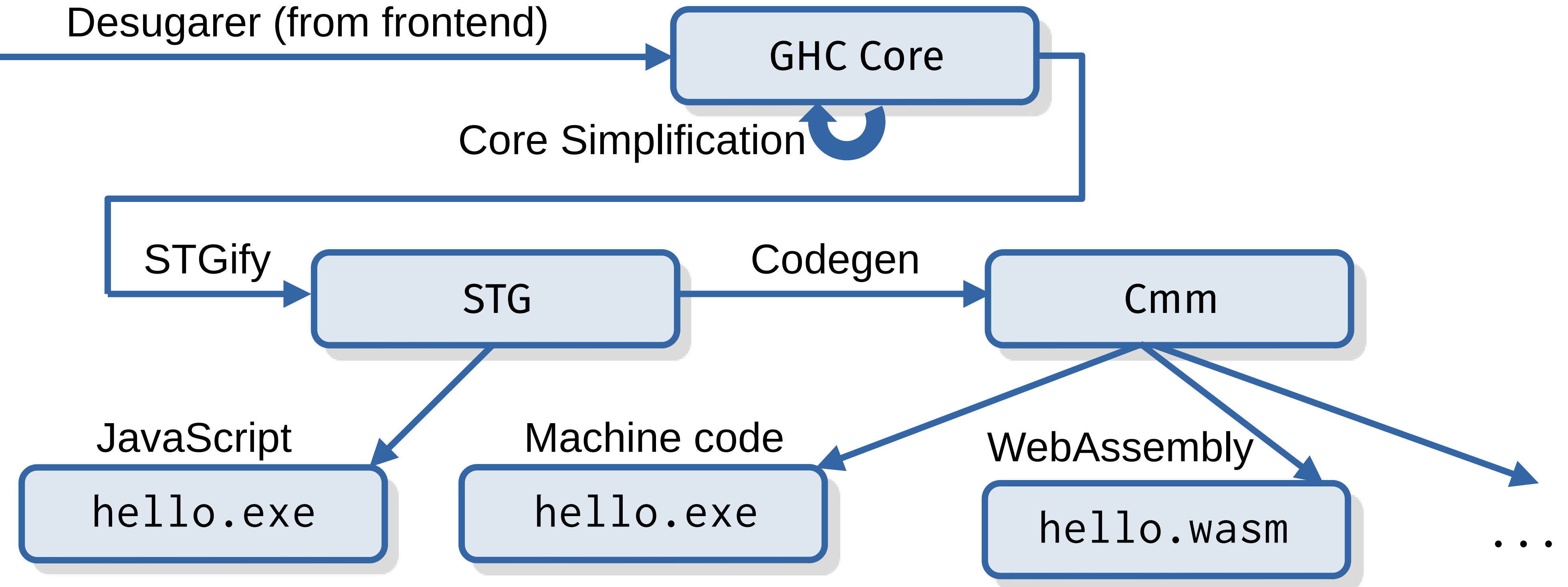
- Key data types: Haskell ASTs in `compiler/Language/Haskell/Syntax`:

```
data HsExpr p
  = HsVar (XVar p) (LIdP p)
  | HsApp (XApp p) (LHsExpr p) (LHsExpr p)
  | HsLam (XLam p)
          (MatchGroup p (LHsExpr p))
  | ...
```

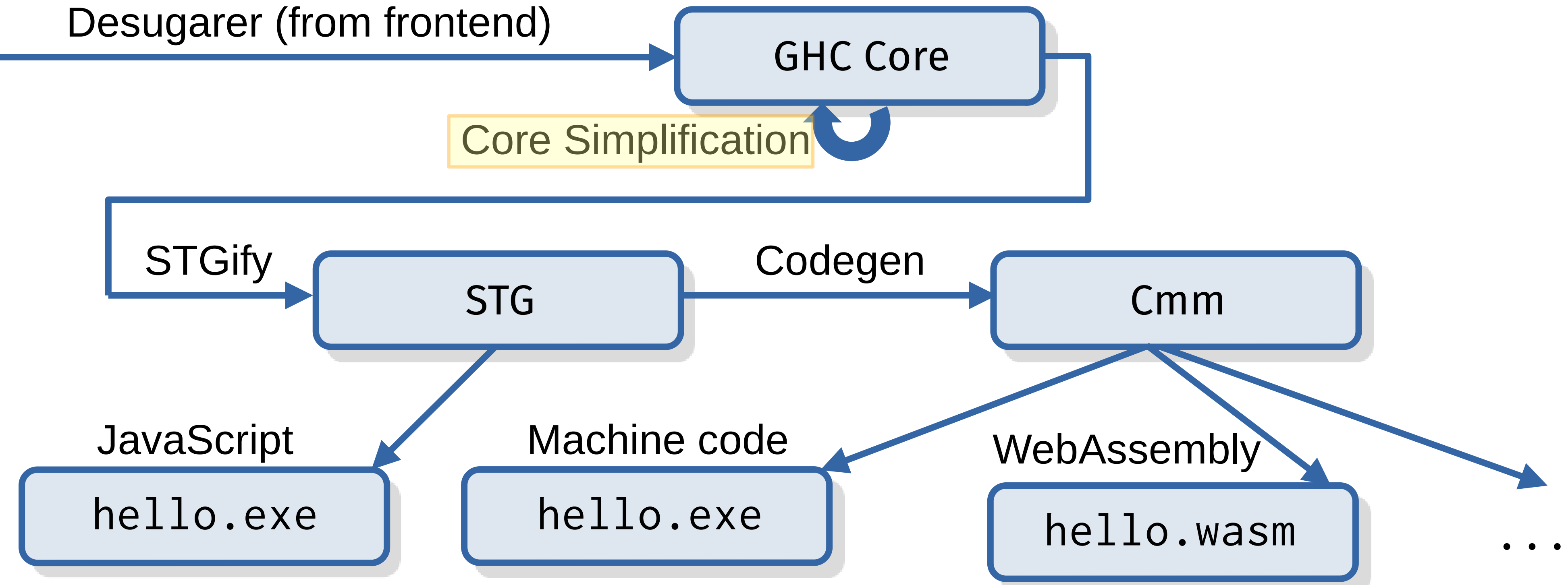
```
data HsType p
  = HsTyVar (XTyVar p)
            PromotionFlag (LIdP p)
  | HsAppTy (XAppTy p)
            (LHsType p) (LHsType p)
  | ...
```

- The `X*` type families are explained in `Note [Trees That Grow]` in `compiler/Language/Haskell/Syntax/Extension.hs`

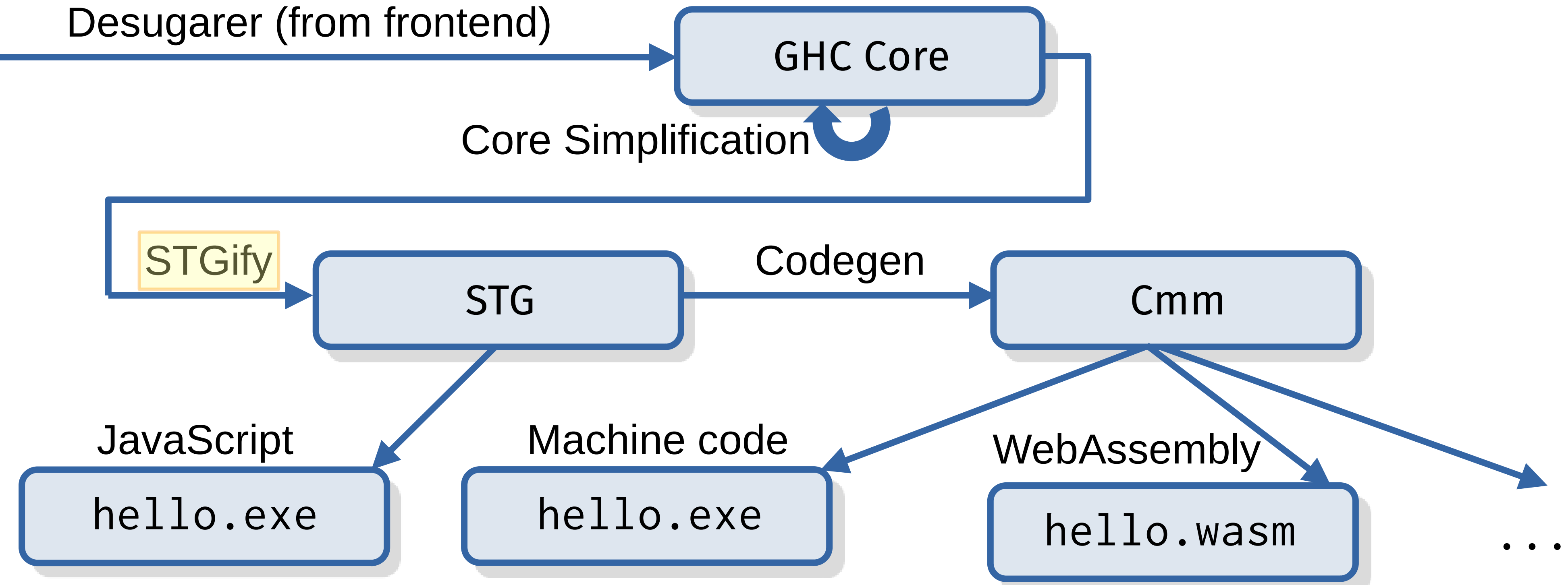
# Compiler pipeline (backend)



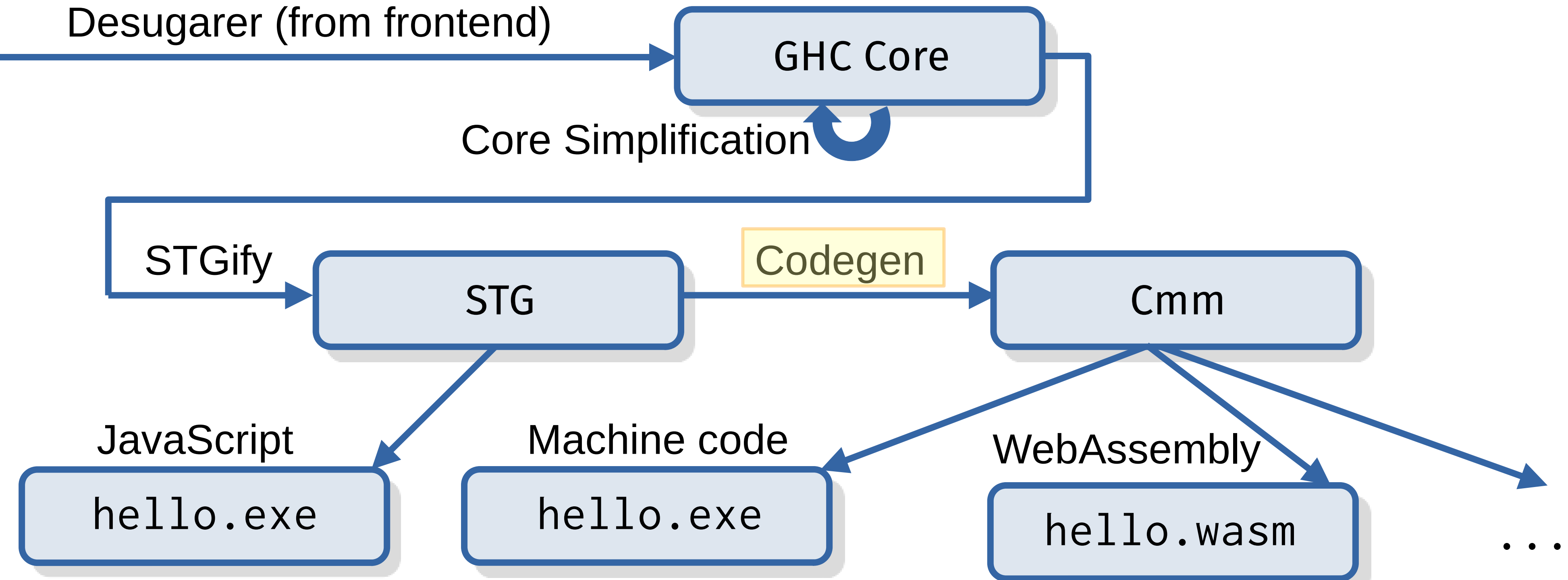
# Compiler pipeline (backend)



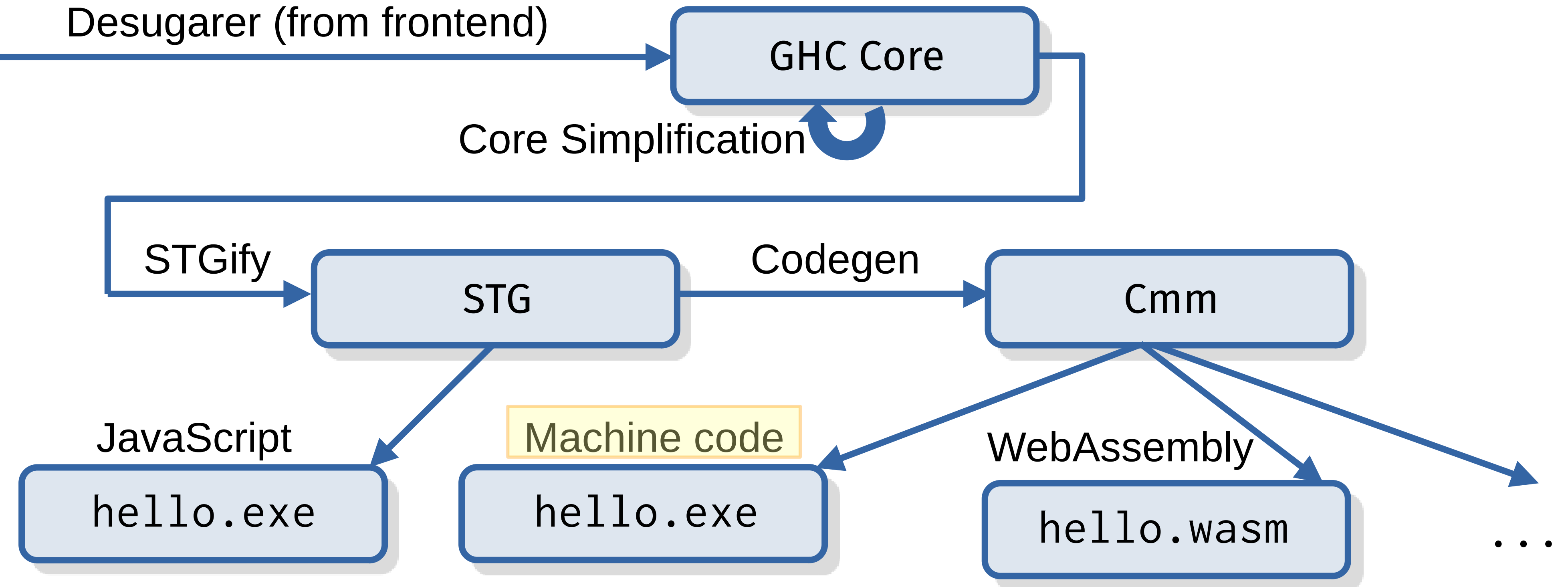
# Compiler pipeline (backend)



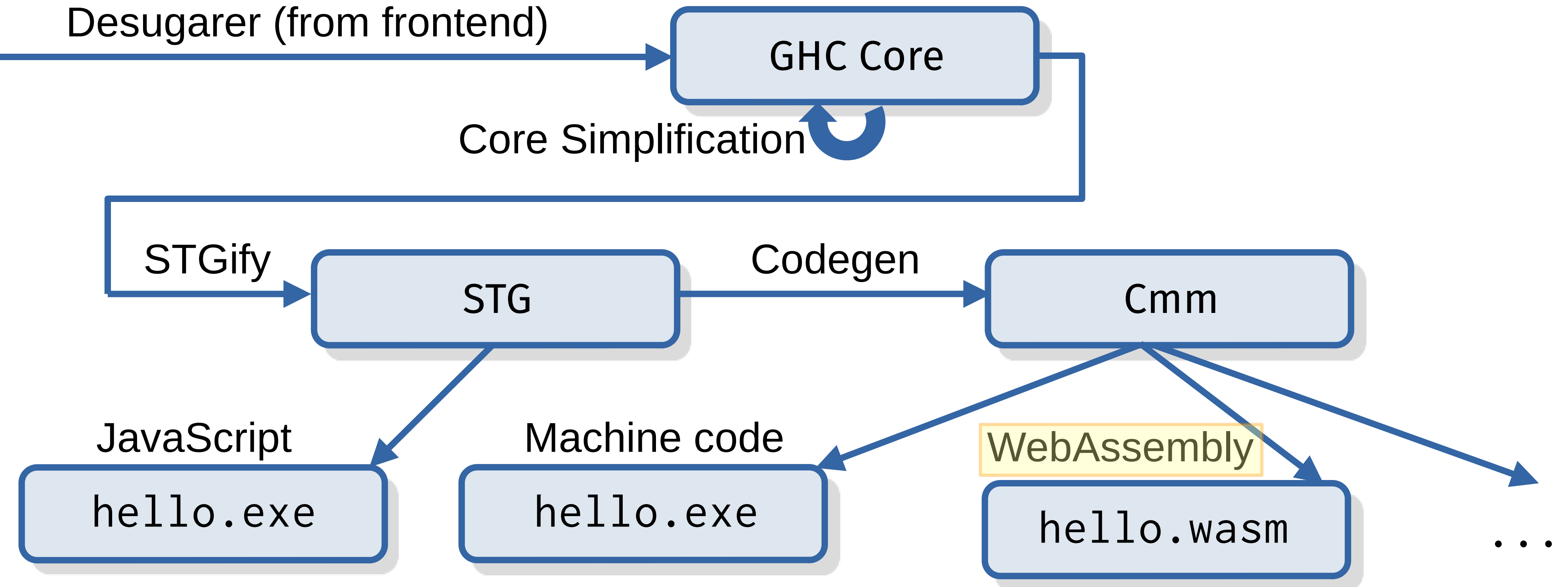
# Compiler pipeline (backend)



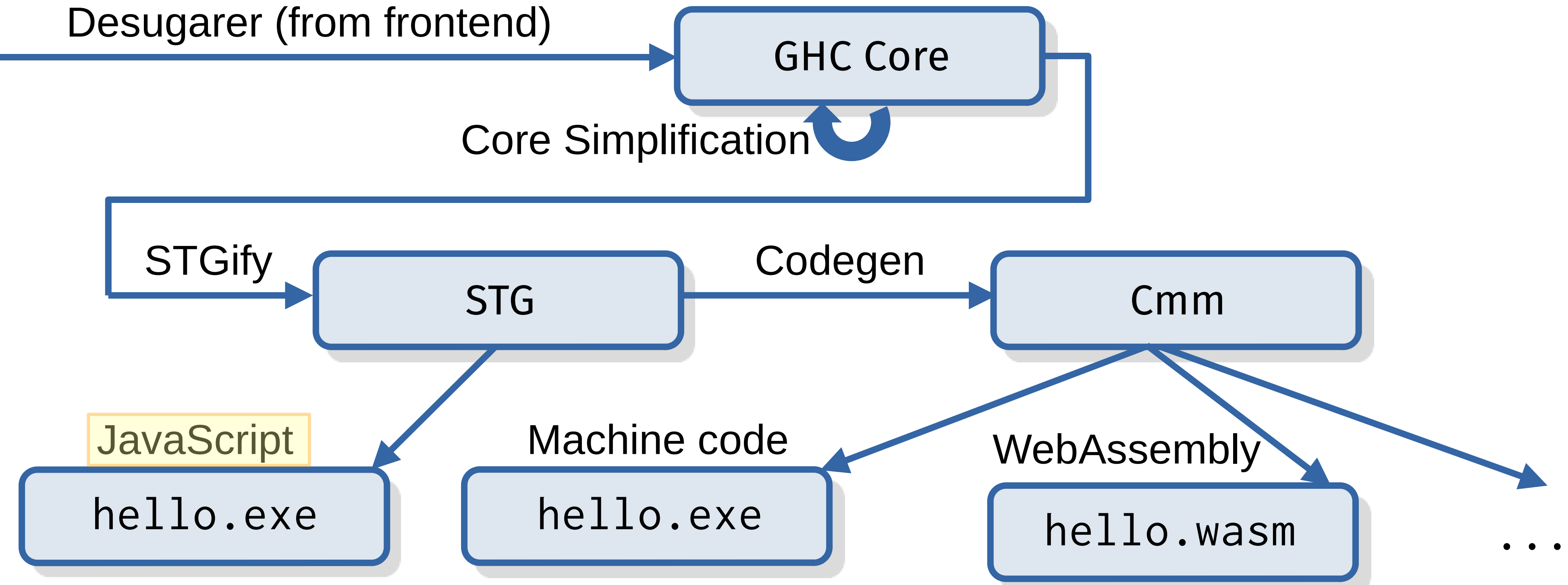
# Compiler pipeline (backend)



# Compiler pipeline (backend)



# Compiler pipeline (backend)



# Compiler pipeline (backend)

- Key data type: GHC Core (in `compiler/GHC/Core.hs`):

# Compiler pipeline (backend)

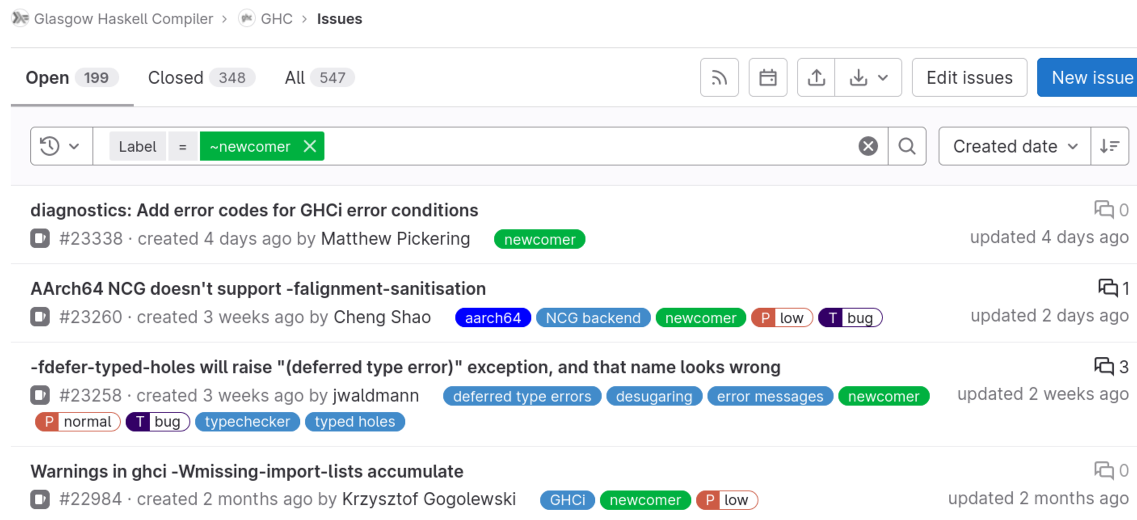
- Key data type: GHC Core (in `compiler/GHC/Core.hs`):

```
data Expr b
  = Var      Id                          -- x
  | Lit      Literal                     -- 27
  | App      (Expr b) (Arg b)            -- e1 e2
  | Lam      b (Expr b)                  -- \x. e
  | Let      (Bind b) (Expr b)           -- let { x = e1; ... } in e2
  | Case     (Expr b) b Type [Alt b] -- case e as x return ty of { ... }
  | Cast     (Expr b) CoercionR          -- e `cast` co
  | Tick     CoreTickish (Expr b)        -- tick<e>
  | Type     Type                        -- ty
  | Coercion Coercion                    -- co
```

# Writing your first patch

# How do I pick an issue to fix?

- Use GitLab's `~newcomer` label:




- Or, just ask one of us!

# The bug-fixing checklist

- Pick a bug, and announce you are working on it
- Add a failing test case
- Fix the bug
- Ensure that the test case now passes
- Refer to issue number in commit message
- Submit GitLab merge request
- Incorporate feedback from reviewers

# The bug-fixing checklist

- Pick a bug, and announce you are working on it
- Add a failing test case
- Fix the bug
- Ensure that the test case now passes
- Refer to issue number in commit message
- Submit GitLab merge request
- Incorporate feedback from reviewers

# The bug-fixing checklist

- Pick a bug, and announce you are working on it
- Add a failing test case
- Fix the bug
- Ensure that the test case now passes
- Refer to issue number in commit message
- Submit GitLab merge request
- Incorporate feedback from reviewers

# The bug-fixing checklist

- Pick a bug, and announce you are working on it
- Add a failing test case
- Fix the bug
- Ensure that the test case now passes
- Refer to issue number in commit message
- Submit GitLab merge request
- Incorporate feedback from reviewers

# The bug-fixing checklist

- Pick a bug, and announce you are working on it
- Add a failing test case
- Fix the bug
- Ensure that the test case now passes
- Refer to issue number in commit message
- Submit GitLab merge request
- Incorporate feedback from reviewers

# The bug-fixing checklist

- Pick a bug, and announce you are working on it
- Add a failing test case
- Fix the bug
- Ensure that the test case now passes
- Refer to issue number in commit message
- Submit GitLab merge request
- Incorporate feedback from reviewers

# The bug-fixing checklist

- Pick a bug, and announce you are working on it
- Add a failing test case
- Fix the bug
- Ensure that the test case now passes
- Refer to issue number in commit message
- Submit GitLab merge request
- Incorporate feedback from reviewers

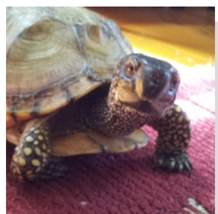
# The bug-fixing checklist

- Pick a bug, and announce you are working on it
- Add a failing test case
- Fix the bug
- Ensure that the test case now passes
- Refer to issue number in commit message
- Submit GitLab merge request
- Incorporate feedback from reviewers

# Review process

Seek approval from at least one person from:

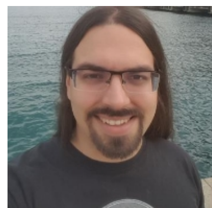
- Codeowners (see CODEOWNERS file)
- GHC maintainers



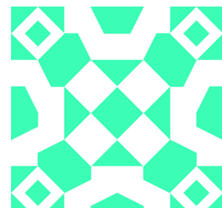
Ben
@bgamari

Matthew
@mpickering

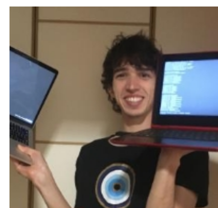
Sam
@sheaf



Andreas
@AndreasK



Zubin
@wz1000



Rodrigo
@alt-romes

# Demo time

https://gitlab.haskell.org/ghc/ghc/-/issues/22559

# Note that...

- There are other ways to contribute besides fixing bugs
  - Documentation fixes
  - Creating minimal examples

# Note that...

- There are other ways to contribute besides fixing bugs
  - Documentation fixes
  - Creating minimal examples

- If you are unsure of how to fix a bug, it can be helpful to ask for help first
  - **At the workshop**: Discord (or talk to one of us!)
  - **Any time**:
    - Ask a question on a GitLab issue
    - IRC: `#ghc` channel on Libera.Chat
    - Matrix: `https://matrix.to/#/#ghc:libera.chat` (bridges with IRC)
    - GHC devs mailing list: `http://www.haskell.org/mailman/listinfo/ghc-devs`

# Questions?