

Если вы видите что-то необычное, просто сообщите мне.

# Contributing to GHC

The Glasgow Haskell Compiler is one of the linchpins of the Haskell community. It is far and away the most widely used compiler for the language. Its many unique features help make Haskell a special language. It also depends a lot of open source contributions! This means anyone can lend a hand and fix issues. This is a great way to give back to the Haskell community and meet some great people along the way!

- [Contributing to GHC 1: Preparation](#)
- [Contributing to GHC 2: Basic Hacking and Organization](#)
- [Contributing to GHC 3: Hacking Syntax and Parsing](#)

# Contributing to GHC 1:

## Preparation

Welcome to our series on Contributing to GHC! The Glasgow Haskell Compiler is perhaps the biggest and most important open source element of the Haskell ecosystem. Without GHC and the hard work that goes into it from many volunteers, Haskell would not be the language it is today. So in this series we'll be exploring the process of building and contributing to GHC. This first part will focus on how to set up our environment for GHC development. If you've already done this, feel free to move onto part 2 of this series, where we'll start some basic hacking!

While many of you are working on Linux or Mac setups, this article uses examples from setting up on Windows. Getting GHC to build on Windows simply involves more hurdles, and we want to show that we can contribute even in the most adverse circumstances. There is a section further down that goes over the most important parts of building for Mac and Linux. You'll generally have to install some of the same packages, but use the native tools on those systems, which are much simpler than on Windows. I'll be following this guide by Simon Peyton Jones, sharing my own complications.

Now, you need to walk before you can run. If you've never used Haskell before, you have to try it out first to understand GHC! Download our Beginner's Checklist to get started! You can also read our Liftoff Series to learn more about the language basics. If you're more interested in applying Haskell than working on GHC, you should read our Haskell Web Series and download our Production Checklist!

## MSYS

The main complication with Windows is that the build tools for GHC are made for Unix-like environments. These tools include programs like `autoconf` and `make`. And they don't work in the normal Windows terminal environment. This means we need some way of emulating a Unix terminal environment in Windows. There are a couple different options for this. One is Cygwin, but

the more supported option for GHC is MSYS 2. So my first step was to install this program. This terminal will apply the "Minimalist GNU for Windows" libraries, abbreviated as "MinGW".

Installing this worked fine the first time. However, there did come a couple points where I decided to nuke everything and start from scratch. Re-installing did bring about one problem I'll share. In a couple circumstances where I decided to start over, I would run the installer, only to find an error stating bash.exe: permission denied. This occurred because the old version of this program was still running on a process. You can delete the process or else just restart your machine to get around this.

Once MSys is working, you'll want to set up your terminal to use MinGW programs by default. To do this, you'll want to set the path to put the mingw directory first:

```
echo "export PATH=/mingw<bitness>/bin:\$PATH" >> ~/.bash_profile
```

Use either 32 or 64 for `<bitness>` depending on your system. Also don't forget the quotation marks around the command itself!

# PACKAGE PREPARATION

Our next step will be to get all the necessary packages for GHC. MSys 2 uses an older package manager called pacman, which operates kind've like apt-get. First you'll need to update your package repository with this command:

```
pacman -Syuu
```

As per the instructions in SPJ's description, you may need to run this a couple times if a connection times out. This happened to me once. Now that pacman is working, you'll need to install a host of programs and libraries that will assist in building GHC:

```
pacman -S --needed git tar bsdtar binutils autoconf make xz \  
curl libtool automake python python2 p7zip patch ca-certificates \  
mingw-w64-$(uname -m)-gcc mingw-w64-$(uname -m)-python3-sphinx \  
mingw-w64-$(uname -m)-tools-git
```

This command typically worked fine for me. The final items we'll need are alex and happy. These are Haskell programs for lexing and parsing. We'll want to install Cabal to do this. First let's set a couple variables for our system:

```
arch=x64_64 # could also be i386
bitness=64  # could also be 32
```

Now we'll get a pre-built GHC binary that we'll use to Bootstrap our own build later:

```
curl -L https://downloads.haskell.org/~ghc/8.2.2/ghc-8.2.2-${arch}-unknown-mingw32.tar.xz | tar -xj -C
/mingw${bitness} --strip-components=1
```

Now we'll use Cabal to get those packages. We'll place them (and Cabal) in /usr/local/bin, so we'll make sure that's created first:

```
mkdir -p /usr/local/bin
curl -L https://www.haskell.org/cabal/release/cabal-install-2.2.0.0/cabal-install-2.2.0.0-${arch}-unknown-
mingw32.zip | bsdtar -xzf- -C /usr/local/bin
```

Now we'll update our Cabal repository and get both alex and happy:

```
cabal update
cabal install -j --prefix=/usr/local/bin alex happy
```

Once while running this command I found that happy failed to install due to an issue with the mtl library. I got errors of this sort when running the ghc-pkg check command:

```
Cannot find any of ["Control\\Monad\\Cont.hi", "Control\\Monad\\Cont.p_hi", "Control\\Monad\\Cont.dyn_hi"]
Cannot find any of ["Control\\Monad\\Cont\\Class.hi", "Control\\Monad\\Cont\\Class.p_hi",
"Control\\Monad\\Cont\\Class.dyn_hi"]
```

I managed to fix this by doing a manual re-install of the mtl package:

```
cabal install -j --prefix=/usr/local/ mtl --reinstall
```

After this step, there were no errors on ghc-pkg check, and I was able to install happy without any problems.

```
cabal install -j --prefix=/usr/local/ happy
Resolving dependencies...
Configuring happy-1.19.9...
Building happy-1.19.9...
Installed happy-1.19.9
```

# GETTING THE SOURCE AND BUILDING

Now our dependencies are all set up, so we can actually go get the source code now! The main workflow for contributing to GHC uses some other tools, but we can start from Github.

```
git clone --recursive git://git.haskell.org/ghc.git
```

Now, you should run the `./boot` command from the `ghc` directory. This resulted in some particularly nasty problems for me thanks to my antivirus. It decided that perl was an existential threat to my system and threw it in the Virus Chest. You might see an error like this:

```
sh: /usr/bin/autoreconf: /usr/bin/perl: bad interpreter: No such file or directory
```

Even after copying another version of perl over to the directory, I saw errors like the following:

```
Could not locate Autom4te/ChannelDefs.pm in @INC (@INC contains /usr/share/autoconf C:/msys64/usr/lib .) at
C:/msys64/usr/bin/autoreconf line 39
```

In reality, the `@INC` path should have a lot more entries than that! It took me a while (and a couple complete do-overs) to figure out that my antivirus was the problem here. Everything worked once I dug perl out of the Virus chest. Once boot runs, you're almost set! You now need to configure everything:

```
./configure --enable-tarballs-autodownload
```

The extra option is only necessary on Windows. Finally you'll use to make command to build everything. Expect this to take a while (12 hours and counting for me!). Once you're familiar with

the codebase, there are a few different ways you can make things build faster. For instance, you can customize the build.mk file in a couple different ways. You can set BuildFlavor=devel2, or you can set stage=2. The latter will skip the first stage of the compiler.

You can also run make from the sub-directories rather than the main directory. This will only build the sub-component, rather than the full compiler. Finally, there's also a make fast command that will skip building a lot of the dependencies.

# MAC AND LINUX

I won't go into depth on the instructions for Mac and Linux here, since I haven't gotten the chance to try them myself. But due to the developer-friendly nature of those systems, they're likely to have fewer hitches than Windows.

On Linux for instance, you can actually do most of the setup by using a Docker container. You'll download the source first, and then you can run this docker command:

```
>> sudo docker run --rm -i -t -v `pwd`:/home/ghc gregweber/ghc-haskell-dev /bin/bash
```

On Mac, you'll need to install some similar programs to windows, but there's no need to use a terminal emulator like MSys. If you have the basic developer tools and a working version of GHC and Cabal already, it might be as simple as:

```
>> brew install autoconf automake  
>> cabal install alex happy haddock  
>> sudo easy_install pip  
>> sudo pip install sphinx
```

For more details, check [here](#). But once you're set up, you'll follow the same boot, configure and make instructions as for Windows.

# CONCLUSION

So that wraps up our first look at GHC. There's plenty of work to do just to get it to build! But you should now move onto part 2, where we'll start looking at some of the simplest modifications we can make to GHC. That way, we can start getting a feel for how the code base works.

If you haven't written Haskell, it's hard to appreciate the brilliance of GHC! Get started by downloading our [Beginners Checklist](#) and reading our [Liftoff Series](#)!

# Contributing to GHC 2: Basic Hacking and Organization

In part 1 of this series, we took our first step into the world of GHC, the Glasgow Haskell Compiler. We summarized the packages and tools we needed to install to get it building. We did this even in the rather hostile environment of Windows. But, at the end of the day, we can now build the project with make and create our local version of GHC.

In this part, we'll establish our development cycle by looking at a very simple change we can make to the compiler. We'll also discuss the architecture of the repository so we'll can make some cooler changes, which you can read about in part 3.

GHC is truly a testament to some of the awesome benefits of open source software. Haskell would not be the same language without it. But to understand GHC, you first have to have a decent grasp of Haskell itself! If you've never written a line of Haskell before, take a look at our Liftoff Series for some tips on how to get going. You can also download our Beginners Checklist.

You may have also heard that while Haskell is a neat language, it's useless from an industry perspective. But if you take a look at our Production Checklist, you'll find tons of tools to write more interesting Haskell programs!

## GETTING STARTED

Let's start off by writing a very simple program in Main.hs.

```
module Main where

main :: IO ()
main = do
    putStrLn "Using GHC!"
```



We can compile this program into an executable using the `ghc` command. We start by running:

```
ghc -o prog Main.hs
```

This creates our executable `prog.exe` (or just `prog` if you're not using Windows). Then we can run it like we can run any kind of program:

```
./prog.exe  
Using GHC!
```

However, this is using the system level GHC we had to install while building it locally!

```
which ghc  
/mingw/bin/ghc
```

When we build GHC, it creates executables for each stage of the compilation process. It produces these in a directory called `ghc/inplace/bin`. So we can create an alias that will simplify things for us. We'll write `lghc` to be a "local GHC" command:

```
alias lghc="~/ghc/inplace/bin/ghc-stage2.exe -o prog"
```

This will enable us to compile our single module program with `lghc Main.hs`.

# HACKING LEVEL 0

Ultimately, we want to be able to verify our changes. So we should be able to modify the compiler, build it again, use it on our program, and then see our changes reflected in the code. One simple way to test the compiler's behavior is to change the error messages. For example, we could try to import a module that doesn't exist:

```
module Main where  
  
import OtherModule (otherModuleString)  
  
main :: IO ()  
main = do
```

```
putStrLn otherModuleString
```

Of course, we'll get an error message:

```
[1 of 1] Compiling Main (Main.hs, Main.o)

Main.hs:3:1: error:
  Could not find module 'OtherModule'
  Use -v to see a list of the files search for.
 |
3 | import OtherModule (otherModuleString)
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Let's try now changing the text of this error message. We can do a quick search for this message in the compiler section of the codebase and find where it's defined:

```
cd ~/ghc/compiler
grep -r "Could not find module" .
./main/Finder.hs:cannotFindModule = cantFindErr (sLit "Could not find module")
```

Let's go ahead and update that string to something a little different:

```
cannotFindModule :: DynFlags -> ModuleName -> FindResult -> SDoc
cannotFindModule = cantFindErr
  (sLit "We were unable to locate the module")
  (sLit "Ambiguous module name")
```

Now let's go ahead and rebuild, except let's use some of the techniques from last week to make the process go a bit faster. First, we'll copy `mk/build.mk.sample` to `mk/build.mk`. We'll uncomment the following line, as per the recommendation from the setup guide:

```
BuildFlavour=devel2
```

We'll also uncomment the line that says `stage=2`. This will restrict the compiler to only building the final stage of the compiler. It will skip past stage 0 and stage 1, which we've already build.

We'll also build from the compiler directory instead of the root ghc directory. Note though that since we've changed our build file, we'll have to boot and configure once again. But after we've re-compiled, we'll now find that we have our new error message!

[1 of 1] Compiling Main (Main.hs, Main.o)

Main.hs:3:1: error:

We were unable to locate the module 'OtherModule'

Use -v to see a list of the files search for.

|

3 |import OtherModule (otherModuleString)

|^^

# GENERAL ARCHITECTURE

Next week, we'll look into making a more sophisticated change to the compiler. But at least now we've validated that we can develop properly. We can make a change, compile in a short amount of time, and then determine that the change has made a difference. But now let's consider the organization of the GHC repository. This will help us think some more about the types of changes we'll make. I'll be drawing on this description written by Simon Peyton Jones and Simon Marlow.

There are three main parts to the GHC codebase. The first of these is the compiler itself. The job of the compiler is to take our Haskell source code and convert it into machine executable code. Here is a very non-exhaustive list of some of the compiler's tasks

1. Determining the location of referenced modules
2. Reading a single source file
3. Breaking that source into its simplest syntactic representation Then there is the boot section. This section deals with the libraries that the compiler itself depends on. They include things such as low level types like `Int` or else `Data.Map`. This section is somewhat more stable, so we won't look at it too much.

The last major section is the Runtime System (RTS). This takes the code generated by the compiler above and determines how to run it. A lot of magic happens in this part that makes Haskell particularly strong at tasks like concurrency and parallelism. It's also where we handle mechanics like garbage collection.

We'll try to spend most of our time in the compiler section. The compilation pipeline has many stages, like type checking and de-sugaring. This will let us zero in on a particular stage and make a

small change. Also the Runtime System is mostly C code, while much of the compiler is in Haskell itself!

# CONCLUSION

That concludes part 2 of our series on GHC. In part 3, we'll take a look at a couple more ways to modify the compiler. After that, we'll start looking at taking real issues from GHC and see what we can do to try and fix them!

If you want to start out your Haskell journey, you should read our Liftoff Series! It will help you learn the basics of this awesome language. For more updates, you can also subscribe to our monthly newsletter!

If you've done some Haskell and are more interested in building apps than working on GHC right away, that's great too! Take a look at our Haskell Web Series for more details.

# Contributing to GHC 3: Hacking Syntax and Parsing

In part 2 of this series, we made more progress in understanding GHC. We got our basic development cycle down and explored the general structure of the code base. We also made the simplest possible change by updating one of the error messages. This week, we'll make some more complex changes to the compiler, showing the ways you can tweak the language. It's unlikely you would make changes like these to fix existing issues. But it'll help us get a better grasp of what's going on. We'll wrap up this series in part 4 by exploring a couple very simple issues.

As always, you can learn more about the basics of Haskell by checking out our other resources. Take a look at our Liftoff Series or download our Beginners Checklist! If you know some Haskell, but aren't ready for the labyrinth of GHC yet, take a look at our Haskell Web Series!

## COMMENTS AND CHANGING THE LEXER

Let's get warmed up with a straightforward change. We'll add some new syntax to allow different kinds of comments. First we have to get a little familiar with the Lexer, defined in `parser/Lexer.x`. Let's try and define it so that we'll be able to use four apostrophes to signify a comment. Here's what this might look like in our code and the error message we'll get if we try to do this right now.

```
module Main where

"" This is our main function
main :: IO ()
main = putStrLn "Hello World!"

...
```

```
Parser error on ``
Character literals may not be empty
|
5 | "" This is our main function
| ^^
```

Now, it's easy enough to add a new line describing what to do with this token. We can follow the example in the Lexer file. Here's where GHC defines a normal single line comment:

```
-- " ~$docsym .* { lineCommentToken }
--" [^$symbol \ ] . * { lineCommentToken }
```

It needs two cases because of Haddock comments. But we don't need to worry about that. We can specify our symbol on one line like so:

```
"""" .* { lineCommentToken }
```

Now we can add the comment above into our code, and it compiles!

# ADDING A NEW KEYWORD

Let's now look at how we could add a new keyword to the language. We'll start with a simple substitution. Suppose we want to use the word `iffy` like we use `if`. Here's what a code snippet would look like, and what the compiler error we get is at first:

```
main :: IO ()
main = do
  i <- read <$> getLine
  iffy i `mod` 2 == 0
    then putStrLn "Hello"
    else putStrLn "World"

...

Main.hs:11:5: error: parse error on input 'then'
|
```

```
11 |   then putStrLn "Hello"
    |   ^^^^
```

Let's do a quick search for where the keyword "if" already exists in the parser section. We'll find two spots. The first is a list of all the reserved words in the language. We can update this by adding our new keyword to the list. We'll look for the reservedIds set in basicTypes/Lexeme.hs, and we can add it:

```
reservedIds :: Set.Set String
reservedIds = Set.fromList [ ...
    , "_" , "iffy" ]
```

Now we also have to parse it so that it maps against a particular token. We can see a line in Lexer.x where this happens:

```
( "if", ITif, 0)
```

We can add another line right below it, matching it to the same ITif token:

```
( "iffy", ITif, 0)
```

Now the lexer matches it against the same token once we start putting the language together. Now our code compiles and produces the expected result!

```
lghc Main.hs
./prog.exe
5
World
```

# REVERSING IF

Now let's add a little twist to this process. We'll add another "if" keyword and call it reverseif. This will change the ordering of the if-statement. So when the boolean is false, our code will execute the first branch instead of the second. We'll need to work a little further upstream. We want to re-use as much of the existing machinery as possible and just reverse our two expressions at the right moment. Let's use the same code as above, except with the reverse keyword. Then if we input 5

we should get Hello instead of World.

```
main :: IO ()
main = do
  i <- read <$> getLine
  reverseif i `mod` 2 == 0
    then putStrLn "Hello"
    else putStrLn "World"
```

So we'll have to start by adding a new constructor to our Token type, under the current if token in the lexer.

```
data Token =
  ...
  | ITif
  | ITreverseif
  ...
```

Now we'll have to add a line to convert our keyword into this kind of token.

```
...
("if", ITif, 0),
("reverseif", ITreverseif, 0),
...
```

As before, we'll also add it to our list of keywords:

```
reservedIds :: Set.Set String
reservedIds = Set.fromList [ ...
  , "_" , "iffy" , "reverseif" ]
```

Let's take a look now at the different places where we use the ITif constructor. Then we can apply them to ITreverseif as well. We can find two more instances in Lexer.x. First, there's the function `maybe_layout`, which dictates if a syntactic construct might need an open brace. Then there's the `isALRopen` function, which tells us if we can start some kind of other indentation. In both of these, we'll follow the example of ITif:

```
maybe_layout :: Token -> P ()
...
```



```

where
  f ITif = pushLexState layout_if
  f ITreverseif = pushLexState layout_if

...
isALRopen ITif = True
isALRopen ITreverseif = True
...

```

There's also a bit in `Parser.y` where we'll need to parse our new token:

```

%token
...
'if' { L_ ITif }
'reverseif' { L_ ITreverseif }

```

Now we need to figure out how these tokens create syntactic constructs. This also seems to occur in `Parser.y`. We can look, for instance, at the section that constructs basic if statements:

```

| 'if' exp optSemi 'then' exp optSemi 'else' exp
  {% checkDoAndIfThenElse $2 (snd $3) $5 (snd $6) $8 >>
    Ams (sLL $1 $> $ mkHsIf $2 $5 $8)
    (mj AnnIf $1:mj AnnThen $4
      :mj AnnElse $7
      :(map (\l -> mj AnnSemi l) (fst $3))
      ++(map (\l -> mj AnnSemi l) (fst $6))) }

```

There's a lot going on here, and we're not going to try to understand it all right now! But there are only two things we'll need to change to make a new rule for `reverseif`. First, we'll obviously need to use that token instead of `if` on the first line.

Second, see that `mkHsIf` statement on the third line? This is where we make the actual Haskell "If" expression in our syntax tree. The `$5` refers to the second instance of `exp` in the token list, and the `$8` refers to the third and final expression. These are, respectively, the True and False branch expressions of our "If" statement. Thus, to reverse our "If", all we need to do is flip this arguments on the third line!

```

| 'reverseif' exp optSemi 'then' exp optSemi 'else' exp
  {% checkDoAndIfThenElse $2 (snd $3) $5 (snd $6) $8 >>

```

```
Ams (sLL $1 $> $ mkHsIf $2 $8 $5)
(mj AnnIf $1:mj AnnThen $4
:mj AnnElse $7
:(map (\l -> mj AnnSemi l) (fst $3))
++(map (\l -> mj AnnSemi l) (fst $6))) }
```

Finally, there's one more change we need to make. Adding this line will introduce a couple new shift/reduce conflicts into our grammar. There are already 233, so we're not going to worry too much about that right now. All we need to do is change the count on the assertion for the number of conflicts:

```
%expect 235 -- shift/reduce conflicts
```

Now when we compile and run our simple program, we'll indeed see that it works as expected!

```
lghc Main.hs
./prog.exe
5
Hello
```

# CONCLUSION

In this part, we saw some more complicated changes to GHC that have tangible effects. In the fourth and final part of this series, we'll wrap up our discussion of GHC by looking at some real issues and contributing via Github.

To learn more about Haskell, you should check out some of our basic materials! If you're a beginner to the language, read our Liftoff Series. It'll teach you how to use Haskell from the ground up. You can also take a look at our Haskell Web Series to see some more advanced and practical skills!