

Navigating unfamiliar code with the Go Guru

Alan Donovan
Go team, Google NYC

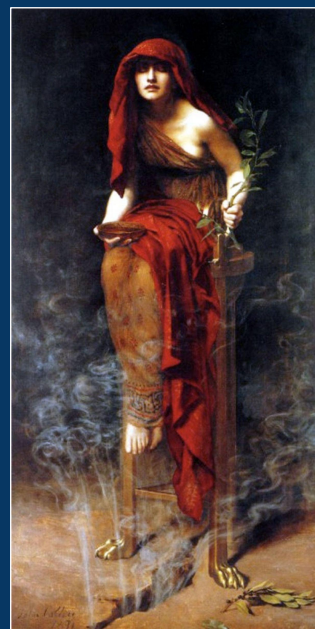
GopherCon 2016
Denver, Colorado
July 11, 2016

guru

“An editor-integrated tool for code navigation and comprehension”

tool golang.org/x/tools/cmd/guru
docs golang.org/s/using-guru

(Until recently, known as *oracle*)



Priestess of Delphi (1891), John Collier

Programmers say they spend their time “writing code”, but most of that time is really spent reading code and making logical deductions about it. Logic is, by definition, mechanical. I believe we should be using machines to help us read, navigate, and understand code, especially unfamiliar code.

This is nothing new for users of IDEs, of course. But I like my current editor (warts and all) and must work in a variety of languages, so a language-specific IDE doesn’t really appeal. The Go community uses a wide (and growing) variety of traditional editors and IDEs.

The goal of *guru* is to provide the services of an IDE, no matter which editor you use. *Guru* is a lightweight headless IDE. But it has some state-of-the-art static analysis features.

The idea for this tool came to me when I was finding my way around google’s internal RPC package, which is surely one of the most complex Go packages there is, with a great many data types, complex aliasing, and a lot of concurrency. As I’ll show you in this talk, *guru* can help you identify the types of any expression and the relationship between channel send and receive statements.

The tool was initially known as *oracle*. The ancient Greeks would bring their tricky questions to the oracle, who would divine the true and often surprising answer by an opaque and inexplicable process. But it turns out that that name was already being used by someone else, so we renamed it *guru*, which still has some of that same

mystique.

Outline

1. Names
2. Types
3. Aliases
4. Future work

In this talk I will give a series of short demonstrations of how the guru tool can answer the kinds of questions that come up all the time during a typical day of coding.

The demos will be interspersed with explanations of how it works.

The talk consists of four parts.

In the first part, we'll see how the answers questions about names.

We'll take a quick interlude to see how the tool interacts with the editor.

In the second part, we'll look at questions about types.

In the third, questions about aliases, that is, the relationships between pointers and the things they point to.

Finally I will conclude with a discussion of future work; there is much still to do.

Names

```
f, err := os.Open("/tmp/foo")
```

```
fmt.Fprintf(f, "hello")
```

```
f.Close()
```

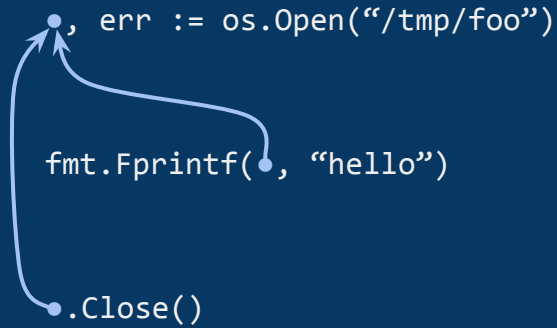
Let's start with names.

So, what's in a name?

A name serves two purposes. The first and most obvious one is documentation. We rightly spent a lot of effort choosing good names, as a good name makes explanatory comments unnecessary.

But the compiler doesn't care about names---and sometimes neither do humans. For example, we use the name 'x' when we have nothing to say. The second and more fundamental purpose of a name is to allow re-use of functions or types we've already defined, or values we've already computed, such as `f` in this example.

Names



That is, names are a *relation*: they relate references to declarations. Names are the way we specify graphs to the compiler.

The first features of the guru tool I'm going to demonstrate help you visualize and navigate this relationship.

Questions about names

Where is this name defined?

Who else refers to this name?

Demo: name queries

1. Identifier highlighting (what)
2. Jump to definition (definition)
3. Find all references (referrers)

1. Identifier highlighting

chat.go, handleConn: alight on various local names

chat.go, main: show resolution of two different 'err' variables

No user action is required.

This is *semantic*, not syntactic highlighting.

2. Jump to definition

Using a keyboard shortcut, I will invoke my editor's "guru: definition" command.

chat.go, handleConn: demonstrate:

 messages (a local var)

 bufio.NewScanner (a qualified identifier)

 input.Scan (a method)

3. Find all references

This time I will invoke the "guru: referrers" command on:

 messages

 net.Listen

Exported names are slower since guru must analyze the whole workspace.

It scans the import declarations and builds a graph, then walks through it in reverse.

The results are a bit like grep, but they're as accurate as a compiler.

Notice that among the results is network.Listen.

It looks like a spurious match, but it's correct because of the renaming import in that file.

Free names

```
func main() {  
    x := 1  
    if err := f(x); err != nil {  
        log.Fatal(err)  
    }  
}  
  
func f(int) error { ... }
```

free names = {x}

The last of the name-related features requires a little explanation.

Often when you're studying a block of code, you want to know what its inputs are. Roughly speaking: if you were to extract the selected statements into their own function, what parameters would that function need?

This set is known as the “free variables” of a block of code, in other words, the set of variables used by the block that are not also defined within it.

(Compilers use the free variables of a function literal to determine what belongs in the closure.)

We can generalize this idea to “free names” by considering constants and types too, though of course types can't be parameters.

In this example, the only free name of the selection is the variable x.

err is not a free variable because it is defined in the block.

Strictly speaking, f is a free function and log is a free package name.

However, for our purposes, we don't consider them free since they are visible throughout the file and so would be accessible to the refactored function.

Sometimes you want this information because you plan to actually extract the logic into its own function.

More often, I don't need to change the code, I just want to understand it.

Guru has a query mode, freevars, that answers exactly this question. Let's see it on a more complex example.

DEMO: \$GOROOT/src/fmt/print.go, (*pp).fmtBytes function, case 'v', 'd'.

Notice how the results are a bit more detailed than just the “free variables” as defined above.

Individual methods and struct fields are broken down, to give a more fine-grained picture.

The results tell us this block of code could be a method of p with two extra parameters.

That's the end of the name-related features.

Behind the scenes

Usage: guru <mode> <position>

mode determines the query and **position** specifies the code of interest

```
$ guru referrers ~/go/src/fmt/print.go:#3669
~/go/src/fmt/print.go:132:6: references to func newPrinter() *pp
~/go/src/fmt/print.go:180.7-180.16:      p := newPrinter()
~/go/src/fmt/print.go:195.7-195.16:      p := newPrinter()
~/go/src/fmt/print.go:214.7-214.16:      p := newPrinter()
~/go/src/fmt/print.go:231.7-231.16:      p := newPrinter()
~/go/src/fmt/print.go:246.7-246.16:      p := newPrinter()
~/go/src/fmt/print.go:263.7-263.16:      p := newPrinter()
```

Wait, what just happened there?

Behind the scenes, the editor fork+exec'd the guru command in a subprocess.

As a user, you never need to think about the command.

It's your editor's responsibility to run the command and display its outputs appropriately.

But it is a command line tool. It takes two arguments. The first, mode, determines the kind of query.

The second is the position, which specifies the current position of the cursor (or selection).

The editor provides the file name and cursor byte offset of the current buffer.

If you have selected a region of text, your editor specifies a pair of byte offsets.

In this example, guru finds all referrers to the identifier at offset 3669 within print.go, which happens to be the function newPrinter.

The output of the tool uses a traditional compiler diagnostic format, with each result appearing on its own line, prefixed by its file name and line and column numbers.

Most editors have great built-in support for displaying output in this format in a useful way, linking each file name to a source location.

As a result, very little logic is required to make most commands work in a new editor.

This simplicity makes it easy to evolve queries or add new ones.

But it's a double-edged sword: queries have (almost) no options, so they have to have good default behaviors.

Query modes

definition	show declaration of selected identifier	Names
referrers	show all refs to entity denoted by selected identifier	
what	show basic information about the selected syntax node	
freevars	show free names of selection	
describe	describe selected syntax: definition, methods, etc	Types
implements	show <i>implements</i> relation for selected type or method	
callees	show possible targets of selected function call	Aliases
callers	show possible callers of selected function	
callstack	show path from callgraph root to selected function	
peers	show send/receive corresponding to selected channel op	
pointsto	show variables the selected pointer may point to	
whicherrs	show possible values of the selected error variable	

This list shows all the query modes that guru currently supports.
The first group relate to names, the second to types, the third to aliasing.
We've seen the first three already and I will present the rest in a moment.

Supported editors

Acme	https://github.com/davidrjenni/A https://github.com/mjibson/aw
Atom.io	https://atom.io/packages/go-oracle
Eclipse	https://github.com/GoClipse/goclipse
Emacs	https://github.com/dominikh/go-mode.el
Sublime Text	https://alvarolm.github.io/GoGuru
Vim	https://github.com/fatih/vim-go
VSCoDe	https://github.com/Microsoft/vscode-go
Minimal support:	ability to run a compiler and display its diagnostics
Fancier:	modified buffer support, highlighting, jump to definition

The Go community has contributed integrations with many popular editors. guru's simple command-line interface reduces the amount of logic required to support Guru in a new editor.

The minimal requirement is the ability to construct a compiler command and display its output in a window, marking up links. Some features require a little more work.

JSON output

```
$ guru -json what ~/go/src/fmt/print.go:#5174
```

```
{
  "enclosing": [...],
  "srcdir": "~/go/src",
  "importpath": "fmt",
  "object": "Fprintf",
  "sameids": [
    "~/go/src/fmt/print.go:179:6",
    "~/go/src/fmt/print.go:190:9"
  ]
}
```

```
// Fprintf formats according to a format
// It returns the number of bytes written
func Fprintf(w io.Writer, format string,
    p := newPrinter()
    p.doPrintf(format, a)
    n, err = w.Write(p.buf)
    p.free()
    return
}

// Printf formats according to a format s
// It returns the number of bytes written
func Printf(format string, a ...interface)
    return Fprintf(os.Stdout, format,
```

source locations
to highlight

Two of the queries I demonstrated, semantic highlighting and jump to definition, were a little unusual.

Both require the editor to interpret the output more closely than it would the output of a typical compiler.

To make this easier, the tool supports JSON output, which almost every editor knows how to parse.

This example shows the query done by the editor when the cursor alights on the identifier `Fprintf` in the `fmt` package.

The **sameids** field reports the set of identifiers that are equivalent to the selected one and that should be highlighted.

This query is called **what**, as in “tell me what I am looking at, as fast as possible”.

Ok, that’s enough about behind the scenes.

Types

The second set of guru features relate to types.

Questions about types

What is the value of this constant?

What is the type of this expression?

What are its operators, fields, and methods?

How much space does this variable occupy in memory?

Which interfaces does this type satisfy?

Obviously, types are a crucial for program comprehension because they describe much of the structure of the program:

They tell us the operators, fields, methods, and size of a value.

They tell us the parameters and results of a function.

And since the type checker must evaluate constants, type information tells us the value of constant expressions.

Adaptations to unavailable type information

```
class C {  
    char *m_pszfoo;  
}
```

“Hungarian” notation

```
Observer observer = new Observer();
```

“stutter”

Historically, there has been a tension between saying more and saying less about types when naming variables.

Microsoft C++ for many years used a convention for variable naming, invented by Charles Simonyi, the designer of MS Word and Excel.

The prefix of each variable name encoded its type.

Instead of `foo`, you would say `m_pszfoo` to mean “a member variable that is a pointer to a nul-terminated string”.

It was called Hungarian notation, after its inventor, and because it looks completely foreign.

In many languages it is common to include a variable’s type in its name: `Observer observer = new Observer();`

These conventions exist because the type deductions made by the compiler are not available to the programmer.

IDEs have rendered them less necessary.

Go takes a stylistic position that short names are good.

(It felt strange at first, but I have come to love it.

Using another language recently I was struck by the sheer quantity of text I had to read and re-read.)

Still, it can sometimes be hard to tell the type of a variable or expression, especially one involving a long chain of selections.

Guru can help.

Demo: type queries

1. Describe selected syntax (`describe`)
2. Show related concrete/interface types (`implements`)

`describe`: (in `server.go`)

- `http.ResponseWriter`: `describe` tells us its methods
- `w.Header()`: a method call; `describe` tells us its type and methods
- `req.URL`: a field; `describe` tells us its fields and methods
- `balmy`: a constant; `describe` shows us its value
- `log`: a package; `describe` shows us its API.

`implements`:

- `ResponseWriter`: two implementations private to the `http` package; also, it implements `io.Writer`
- `Celsius`: implements `fmt.Stringer`.
- `handler`: implements `http.Handler`

Aliases

The third and final set of guru features relate to aliasing.

What is aliasing?

```
var x int
p := &x
print(x) // "0"
*p = 1
print(x) // "1"
```

An assignment to `*p` changes the value of `x`, and vice versa

`*p` and `x` are *aliases* for the same variable

Pointers are not the only reference type in Go:

```
*T    []T    map[K]V    chan T    func
```

What is aliasing?

“Aliasing” is when two different expressions denote the same variable.

Aliasing is commonplace in all ALGOL-like languages.

No matter which alias (`*p` or `x`) is used in an assignment to a variable, the effect of the update is observed by all of them.

Aliasing complicates program comprehension because it makes it hard to see the statements that update a variable.

But there are static analysis techniques that can help.

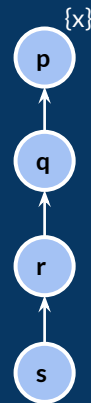
Pointer analysis

golang.org/x/tools/go/pointer

A whole-program static analysis that tells us:
which variables might this pointer point to?

```
var x int
p := &x
s := f(p)

func f(q *int) (r *int) { return q }
```



We use pointer analysis to answer the question: *which variables might this pointer point to?*

This is the basic idea:

The analysis inspects every statement in the program and builds a graph.

Nodes in the graph are pointer values.

Edges are assignment statements.

Associated with each node is a set, the “points to” set.

Initially, these sets represent the variables whose address is taken. For example, p points to x.

From this graph, we can deduce what any pointer points to by computing the transitive closure.

For example, p, q, r, and s all may point to x.

I’m simplifying a lot here. Pointer analysis is a complex topic.

To find out more, watch my GothamGo NYC or dive into the package documentation for golang.org/x/tools/go/pointer.

Questions about aliases

Which variable does this pointer point to?

Which kinds of error might this err variable contain?

Where might this dynamic call dispatch to?

Where is this function called from?

By what path might this function be called from main?

Where is the corresponding receive for this channel send?

All of these queries require, in general, an understanding of aliasing.

The first group of questions are directly about variables and what they point to.

The second group of questions are about the call graph.

You see, 'func' values are references to functions, and interfaces are (conceptually) references to concrete types, so a pointer analysis can tell us the possible targets of a dynamic function or method call.

The final question is about channels. Pointer analysis helps us here too.

Pointer analysis tells us channel peers too


```
M: ch := make(chan int, 1)
    ch1 := ch
```

```
S: ch1 <- 1
    ch2 := ch
```

```
R: print(<-ch2)
```

```
pointsto(ch1) = {M}
pointsto(ch2) = {M}
```

Because these sets intersect,
that is, ch1 and ch2 may alias,
the operations S and R may communicate



When reading a concurrent Go program, it's natural to want to follow a value sent into a channel to see where it pops out at the other end.

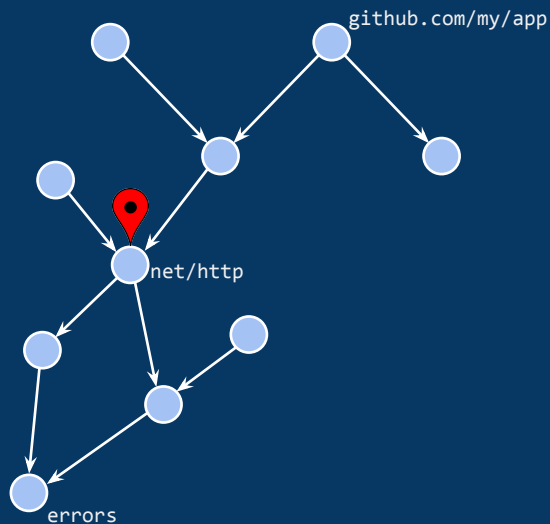
But because a channel may have many aliases, such as ch1 and ch2 in this example, it can be hard to see.

Often all we have to go on is the element type, and there may be many channels of common types like int and bool.

Pointer analysis tells us that both of these channel values, ch1 and ch2, may alias the same channel object, M.

This means that the send S and receive R may communicate with each other.

Pointer analysis scope



Consider a query about the net/http package

I want to quickly explain an important concept about the pointer-analysis based queries:

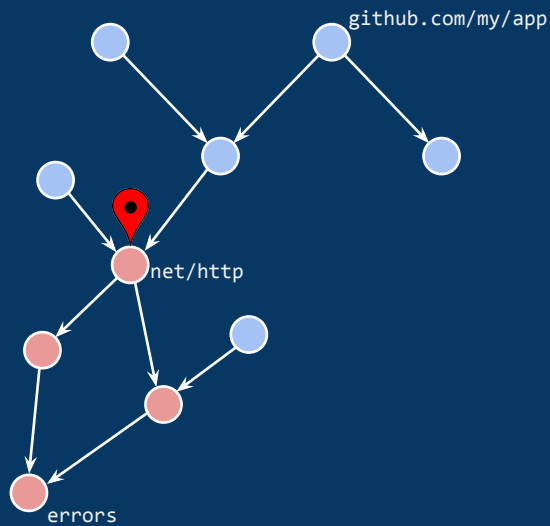
I said before that the guru tool needs no configuration (besides \$GOPATH).

Actually, the pointer analysis needs one extra piece of information: the set of main (or test) packages it should analyze.

This is called the “pointer analysis scope”.

This figure shows the packages in your workspace. Consider a query about this package, `net/http`.

Forward analysis

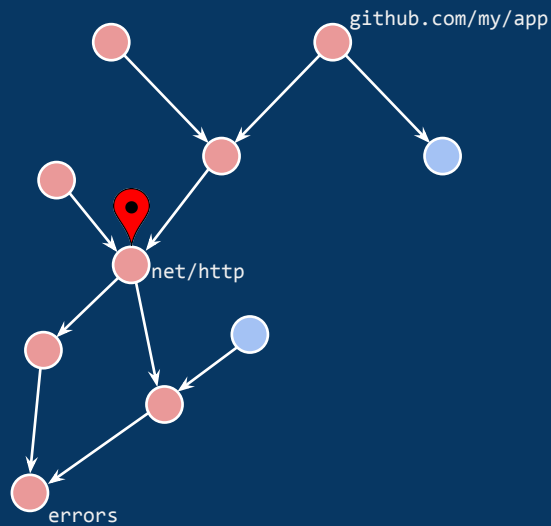


A forward analysis works
forwards from the query package
Example: describe, definition

A forward analysis, such as *describe* or *definition*, must inspect all the packages forward of the query point.

The cost of the query is proportional to the size of the program. (You can think of compilation as a forward analysis.)

Backward analysis



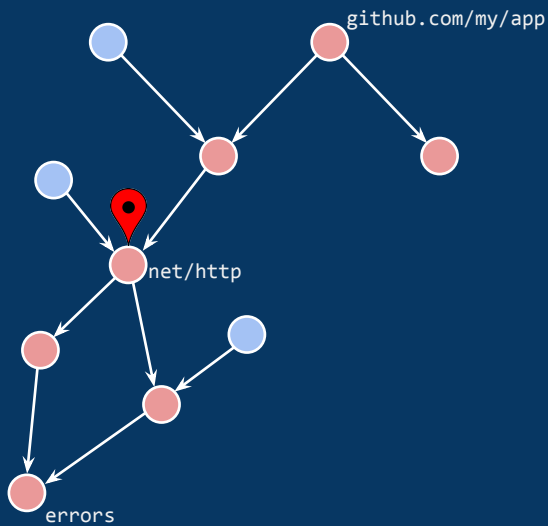
A backward analysis also works
backwards from the query package
Example: `referrers`, `implements`

In contrast, a backward analysis, such as `referrers`, must *additionally* inspect all the packages that depend on the the query package.
The cost of the query may be proportional to the size of the entire workspace.

Pointer analysis

```
with scope {github.com/my/app}
```

Pointer analysis works forwards from the *scope*, a set of main packages specified by the user



Pointer analysis is conceptually a backwards analysis, but it's simply too expensive for the entire workspace.

So, the user must tell the tool the set of packages they are interested in.

We call this set of packages the *pointer analysis scope*.

(Perhaps it's the application you're currently working on, or a client and server.)

They must be main (or test) packages, because it's a whole-program analysis.

The pointer analysis then analyzes forwards from those packages.

Remember: to use the pointer analysis features, you must set the scope.

More on this later.

Demo: alias queries

1. Points-to analysis (`pointsto`, `whicherrs`)
2. Channel send/receive peers (`peers`)
3. Call graph queries (`callers`, `callees`, `callstack`)

Demo (chat.go)

`pointsto`: `w`, an interface, may have two concrete dynamic types. Each is a pointer. The query shows us where the object is allocated.

`whicherrs`: on the error returned by `http.ListenAndServe`, tells us three possible errors.

`peers`: on `ch` in `handleConn`, shows us all the operations that use this channel

`callers`: on `Celsius.String`, shows us the dynamic call from `fmt`.

`callees`: on `w.Header()`, shows the two possible concrete `Header` methods.

`callstack`: on `handleRoot`, shows us a path from `main`, through the HTTP server, to the handler. Notice one call creates a new goroutine.

Future work

Finally, I'm going to discuss some remaining challenges.

Challenge: stateless design

Pro: Simple implementation
Results always fresh

Con: Slower for larger code bases

Solution: Disk-based cache of saved type information
(Go 1.7 export data format contains position information)
In-memory cache of workspace import graph
Fsnotify-based daemon responds to file system operations

Sacrifice some freshness for better performance

Proposed changes will not affect the command-line interface

A goal of the current design was to make the tool completely stateless and to use the source on disk (or modified in the editor buffer) as the ground truth. This does simplify the implementation, but I recognize that for users working in very large code bases, some queries can be too slow.

The external Go programs I work on rarely exceed 200,000 lines, and guru works great at that scale, but CockroachDB, for example, pulls in about two million lines of code.

The forward queries are still quite fast, but the reverse queries may take several seconds, and the pointer analysis queries may take tens of seconds.

Currently, guru uses the PTA scope to bound other reverse analyses, but this is really just a stopgap.

I plan to make the tool stateful, using a disk-based database of compiled type information,

and a long-lived “indexer” portion that maintains an in-memory cache of the package import graph.

Go 1.8 will feature a new export data format for serialized type information;

I made sure that this format includes source position information. (Before, it was useless to guru.)

The “indexer” will respond to file system notifications and update these structures.

This design will make Name and Type-based queries very fast and give partial results to some of the call graph related queries.

However, some query results may lag several seconds behind recent edits

Incidentally, the very narrow interface between the editor and the guru command enables radical internal design changes without changes to the editor-specific code.

Challenge: pointer analysis

Pro:	Impressively precise results
Con:	Requires extra configuration (“scope”) Requires input that compiles
Solution:	Use simpler analysis if pointer analysis unavailable (call graph queries only)
Con:	Can be slow
Solution:	Run pointer analysis asynchronously, with option to refresh Optimize <code>set<int></code> representation further

Pointer analysis is a tricky thing.

On the one hand, it delivers some of the most impressive results of the tool.

On the other, it is somewhat slow, requires configuration (the “scope”), and requires a program that compiles.

These characteristics make it less than ideal for an interactive app.

(It’s a much better fit for committed code. I mapreduce it over Google’s multimillion line Go codebase each night.)

Again, the solution is offline computation:

Run the pointer analysis asynchronously and dump its solution to a database.

Then query that database, even if it is slightly stale, with the option to trigger a refresh.

In addition, some queries like callers/callees can be made to deliver partial results without pointer analysis.

Thanks to:

Guru tool:

Robert Griesemer
David Crawshaw
Michael Matloob

Dominik Honnef
Daniel Morsing

Editor integration:

Acme	David R Jenni, Matt Jibson
Atom.io	Scott Baron
Eclipse	Bruno Medeiros
Emacs	Dominik Honnef
Vim	Fatih Arslan
Sublime	Alvaro M, Hewei Liu
	Dan Mace, Jesse Meek
VSCode	Luke Hoban

golang.org/x/tools/cmd/guru

Thanks to my colleagues at Google, who reviewed guru and the many libraries it depends on;
external contributors and reviewers;
and the many people who have integrated guru with an editor.