Given the following code in go:

```
func (entry *LogfileEntry) FindLogIndex(logR io.ReadSeeker) (n int, err error) {
	var currPos int64
	currPos, err = logR.Seek(LogStart, io.SeekCurrent)
	if err != nil {
		return 0, err
	}
	for entry.WriteCount - n >= 0 {
		line, err := br.ReadString('\n')
		if err != nil && err != io.EOF {
			return 0, err
		}
		if err == io.EOF {
			break
		}
...
}
```

almost every function call return err value is checked and bails if is not nil (or maybe looks at the kind of error and then bails).

There have been some proposals to try to declutter this kind of code (there may have been more I am not aware of), see:

https://github.com/golang/go/issues/31442
https://github.com/golang/go/issues/32437
https://go.googlesource.com/proposal/+/master/design/go2draft-error-handling-overview.md

This is yet another one of those proposals, hopefully without most of their drawbacks.

We propose two reserved words, **inject** and **eject** which, together, constitute a general bailing mechanism for functions.

Inject works on a function call. We say the function being called is being injected into this function.

For a function to be injected:

- The injected function must return the same types (or compatible types) as the function from which it is called.
- The injected function must contain at least one eject statement.

Additionally,

- A function containing an eject statement can only be injected, it cannot be called regularly.

If any of these conditions are not met, a compilation error is generated.

The eject reserved word functions as return, except it provokes another return at the injection site.

A complete example with general error bailing functions equivalent to the above would be (note that both bailing functions are general and can be reused):

```
func ErrBail[T any](v T, err error) (t T, err error) {
	if err != nil {
		eject t, err
	}
	return t, err
}

func NoEofBail[T any](v T, err error) (t T, err error) {
	if err != nil && err != io.EOF {
		eject t, err
	}
	return t, err
}

func (entry *LogfileEntry) FindLogIndex(logR io.ReadSeeker) (n int, err error) {
	var currPos int64
	currPos, err = logR.Seek(LogStart, io.SeekCurrent)
	inject ErrBail(currPos, err)
	for entry.WriteCount - n >= 0 {
		line, err := br.ReadString('\n')
		inject NoEofBail(line, err)
		if err == io.EOF {
			break
		}
...
}
```

A simpler less general no generics bailing function which also works for the above example would be:

```
func ErrBail(n int, err error) (n int, err error) {
	if err != nil {
		eject n, err
	}
	return n, err
}
```

Scopes and everything else work as usual. In fact, most of it is syntactic sugar, i.e. it can be implemented by rewriting the code:

```
inject NoEofBail(line, err)
```

could be rewritten by the compiler under the covers to be:

```
if nx, errx, iseject := NoEofBail(line, err); iseject {
	return nx, errx
}
```

and the injected function would be changed to:

```
func ErrBail(n int, err error) (n int, err error, iseject bool) {
	if err != nil {
		return n, err, true
	}
	return n, err, false
}
```

The mechanism is general enough that one can add context or wrap the error with a new error or do whatever was being done before as needed; this is a completely general mechanism:

```
func ErrBailMsg(n int, msg string, err error) (n int, err error) {
	err = fmt.Errorf("[%d] %s: %s\n", n, msg, err)
	if err != nil && n > 0 {
		eject n, err
	}
	return n, err
}
```

For now, the return values of an injected function when it is not ejecting are ignored (see that the variables in the rewrite example are local to the if statement).
This could be extended later if there is some use/need for it, so that one can inject an assignment with a function call as rvalue.

Note that, if bailing functions are small, they are prime candidates for inlining and the code needs not be any slower than before: in fact, the final generated code after optimization can be exactly the same.