Build is composed of a bunch of parts; a lexer, parser, preprocessor, processor, and post-processor to be exact. All these are part of what I call the builder (as in graph builder) which, you guessed it builds the build graph.

Lexer

Lexer is not really that sophisticated in terms of what I do with it, practically I've seen Rob's lexical scanning in go.

Parser

Parser get's all these tokens and turns then in to a function and some variables. A file's AST representation that looks like this

```
type File struct {
        Path  string
        Funcs []###Func
        Vars  map[string]interface{}
}
```

PreProcessor

At this point in the lifecycle of a build file, we have a rudimentary AST so we can do things like check if we have duplicate load functions in a file.

```
func processDupeLoad(f *ast.File) error {
        seenFile := make(map[string]*ast.Func)
        for _, function := range f.Funcs {
                if function.Name != "load" {
                        continue
                }
                var fileName string
                switch function.AnonParams[0].(type) {
                case string:
                        fileName = function.AnonParams[0].(string)
                default:
                        errorMessage := `load must always be in this form `load("//foo/bar/B
                        log.Fatal(errorMessage)
                }

                if before, seen := seenFile[fileName]; seen {
                        return fmt.Errorf("`load` function in file %s, loads from same file
                                filepath.Join(f.Path, function.File),
                                function.AnonParams[0].(string),
                                function.Line,
                                before.Line,
                        )
                } else {
                        seenFile[fileName] = function
                }
        }
        return nil
}
```

2016/01/09 17:34:18 error processing document: 'load' function in file /Users/sevki/Code/harvey/sys/src/libthread/BUILD, loads from same file //sys/src/FLAGS twice. try merging load functions on line 2 and 1.

We can also check for other stuff, like duplicate functions that describe the same target and so on.

Processor

At this stage we are still dealing with an AST, processor is what takes all the ast.Func stuff and returns their respective graph objects.

There are two types of functions essentially, those who return and those that become build targets. Functions like **glob** , **load** , **version** get processed right inside processor.

Caveat: Files that are added trough the **glob** function are an exception to the abosluting mechanism in the post processor.

All the non returning functions are then valmorphanize in to their respective target interfaces and that's how they will spend the rest of their lives.

PostProcessor

At this stage all the targets are in their go struct forms. Last two things build does is process the dependencies (which is a pretty straight forward thing to do targets that start with : get the path attached to them) and processing the paths, which involves more work.

For instance the **syn** target in harvey installs a **x.tab.c** file, which **rc** lists in it's **srcs** field, but because it is also in the map of things that are installed from the **syn** target the post processor doesn't absolute the path.

How is this ok? Doesn't build take hash of the files?

Yes, but a nodes hash is determined by it's dependencies so if the file that is produced by the target has changed, so should the hash of the dependency node and every other node that depends on it.