

## Decorated Syntax Tree

The `dst` package enables manipulation of a Go syntax tree with high fidelity. Decorations (e.g. comments and line spacing) remain attached to the correct nodes as the tree is modified.

### Where does go/ast break?

The `go/ast` package wasn't created with source manipulation as an intended use-case. Comments are stored by their byte offset instead of attached to nodes, so re-arranging nodes breaks the output. See this Go issue for more information.

Consider this example where we want to reverse the order of the two statements. As you can see the comments don't remain attached to the correct nodes:

```
1  code := `package a
2
3  func main(){
4      var a int    // foo
5      var b string // bar
6  }
7  `
8  fset := token.NewFileSet()
9  f, err := parser.ParseFile(fset, "", code, parser.ParseComments)
10 if err != nil {
11     panic(err)
12 }
13
14 list := f.Decls[0].(*ast.FuncDecl).Body.List
15 list[0], list[1] = list[1], list[0]
16
17 if err := format.Node(os.Stdout, fset, f); err != nil {
18     panic(err)
19 }
20
21 //Output:
22 //package a
23 //
24 //func main() {
25 //    // foo
26 //    var b string
27 //    var a int
28 //    // bar
29 //}
```

---

Here's the same example using `dst`:

```
1 code := `package a
2
3 func main(){
4     var a int    // foo
5     var b string // bar
6 }
7 `
8 f, err := decorator.Parse(code)
9 if err != nil {
10     panic(err)
11 }
12
13 list := f.Decls[0].(*dst.FuncDecl).Body.List
14 list[0], list[1] = list[1], list[0]
15
16 if err := decorator.Print(f); err != nil {
17     panic(err)
18 }
19
20 //Output:
21 //package a
22 //
23 //func main() {
24 //    var b string // bar
25 //    var a int    // foo
26 //}
```

## Usage

Parsing a source file to `dst` and printing the results after modification can be accomplished with several `Parse` and `Print` convenience functions in the `decorator` package.

For more fine-grained control you can use `Decorator` to convert from `ast` to `dst`, and `Restorer` to convert back again.

## Comments

Comments are added at decoration attachment points. See [here](#) for a full list of these points, along with demonstration code of where they are rendered in the output.

The decoration attachment points have convenience functions `Append`, `Prepend`, `Replace`, `Clear` and `All` to accomplish common tasks. Use the full text of your comment including the `//` or `/**/` markers. When adding a line comment, a newline is automatically rendered.

---

```

1 code := `package main
2
3 func main() {
4     println("Hello World!")
5 }`
6 f, err := decorator.Parse(code)
7 if err != nil {
8     panic(err)
9 }
10
11 call := f.Decls[0].(*dst.FuncDecl).Body.List[0].(*dst.ExprStmt).X.(*dst
    .CallExpr)
12
13 call.Decls.Start.Append("// you can add comments at the start...")
14 call.Decls.Fun.Append("/* ...in the middle... */")
15 call.Decls.End.Append("// or at the end.")
16
17 if err := decorator.Print(f); err != nil {
18     panic(err)
19 }
20
21 //Output:
22 //package main
23 //
24 //func main() {
25 //    // you can add comments at the start...
26 //    println /* ...in the middle... */ ("Hello World!") // or at the end
27 //}

```

## Spacing

The **Before** property marks the node as having a line space (new line or empty line) before the node. These spaces are rendered before any decorations attached to the **Start** decoration point. The **After** property is similar but rendered after the node (and after any **End** decorations).

```

1 code := `package main
2
3 func main() {
4     println(a, b, c)
5 }`
6 f, err := decorator.Parse(code)
7 if err != nil {
8     panic(err)
9 }
10
11 call := f.Decls[0].(*dst.FuncDecl).Body.List[0].(*dst.ExprStmt).X.(*dst
    .CallExpr)

```

---

```

12
13 call.Decs.Before = dst.EmptyLine
14 call.Decs.After = dst.EmptyLine
15
16 for _, v := range call.Args {
17     v := v.(*dst.Ident)
18     v.Decs.Before = dst.NewLine
19     v.Decs.After = dst.NewLine
20 }
21
22 if err := decorator.Print(f); err != nil {
23     panic(err)
24 }
25
26 //Output:
27 //package main
28 //
29 //func main() {
30 //
31 //     println(
32 //         a,
33 //         b,
34 //         c,
35 //     )
36 //
37 //}

```

## Decorations

The common decoration properties ([Start](#), [End](#), [Before](#) and [After](#)) occur on all nodes, and can be accessed with the [Decorations\(\)](#) method on the [Node](#) interface:

```

1 code := `package main
2
3 func main() {
4     var i int
5     i++
6     println(i)
7 }`
8 f, err := decorator.Parse(code)
9 if err != nil {
10     panic(err)
11 }
12
13 list := f.Decls[0].(*dst.FuncDecl).Body.List
14
15 list[0].Decorations().Before = dst.EmptyLine
16 list[0].Decorations().End.Append("// the Decorations method allows
    access to the common")

```

---

```
17 list[1].Decorations().End.Append("// decoration properties (Before,
    Start, End and After)")
18 list[2].Decorations().End.Append("// for all nodes.")
19 list[2].Decorations().After = dst.EmptyLine
20
21 if err := decorator.Print(f); err != nil {
22     panic(err)
23 }
24
25 //Output:
26 //package main
27 //
28 //func main() {
29 //
30 //     var i int // the Decorations method allows access to the common
31 //     i++      // decoration properties (Before, Start, End and After)
32 //     println(i) // for all nodes.
33 //
34 //}
```

**dstutil.Decorations** While debugging, it is often useful to have a list of all decorations attached to a node. The dstutil package provides a helper function `Decorations` which returns a list of the attachment points and all decorations for any node:

```
1 code := `package main
2
3 // main comment
4 // is multi line
5 func main() {
6
7     if true {
8
9         // foo
10        println( /* foo inline */ "foo")
11    } else if false {
12        println /* bar inline */ ("bar")
13
14        // bar after
15
16    } else {
17        // empty block
18    }
19 }`
20
21 f, err := decorator.Parse(code)
22 if err != nil {
23     panic(err)
24 }
25
```

---

```

26 dst.Inspect(f, func(node dst.Node) bool {
27     if node == nil {
28         return false
29     }
30     before, after, points := dstutil.Decorations(node)
31     var info string
32     if before != dst.None {
33         info += fmt.Sprintf("- Before: %s\n", before)
34     }
35     for _, point := range points {
36         if len(point.Decs) == 0 {
37             continue
38         }
39         info += fmt.Sprintf("- %s: [", point.Name)
40         for i, dec := range point.Decs {
41             if i > 0 {
42                 info += ", "
43             }
44             info += fmt.Sprintf("%q", dec)
45         }
46         info += "]\n"
47     }
48     if after != dst.None {
49         info += fmt.Sprintf("- After: %s\n", after)
50     }
51     if info != "" {
52         fmt.Printf("%T\n%s\n", node, info)
53     }
54     return true
55 })
56
57 //Output:
58 //*dst.FuncDecl
59 //- Before: NewLine
60 //- Start: ["// main comment", "// is multi line"]
61 //
62 //*dst.IfStmt
63 //- Before: NewLine
64 //- After: NewLine
65 //
66 //*dst.ExprStmt
67 //- Before: NewLine
68 //- Start: ["// foo"]
69 //- After: NewLine
70 //
71 //*dst.CallExpr
72 //- Lparen: ["/* foo inline */"]
73 //
74 //*dst.ExprStmt
75 //- Before: NewLine
76 //- End: ["\n", "\n", "// bar after"]

```

---

---

```
77 //- After: NewLine
78 //
79 //*dst.CallExpr
80 //- Fun: ["/* bar inline */"]
81 //
82 //*dst.BlockStmt
83 //- Lbrace: ["\n", "// empty block"]
```

## Newlines

The `Before` and `After` properties cover the majority of cases, but occasionally a newline needs to be rendered inside a node. Simply add a `\n` decoration to accomplish this.

## Clone

Re-using an existing node elsewhere in the tree will panic when the tree is restored to `ast`. Instead, use the `Clone` function to make a deep copy of the node before re-use:

```
1 code := `package main
2
3 var i /* a */ int`
4
5 f, err := decorator.Parse(code)
6 if err != nil {
7     panic(err)
8 }
9
10 cloned := dst.Clone(f.Decls[0]).(*dst.GenDecl)
11
12 cloned.Decls.Before = dst.NewLine
13 cloned.Specs[0].(*dst.ValueSpec).Names[0].Name = "j"
14 cloned.Specs[0].(*dst.ValueSpec).Names[0].Decls.End.Replace("/* b */")
15
16 f.Decls = append(f.Decls, cloned)
17
18 if err := decorator.Print(f); err != nil {
19     panic(err)
20 }
21
22 //Output:
23 //package main
24 //
25 //var i /* a */ int
26 //var j /* b */ int
```

---

## Apply

The `dstutil` package is a fork of [golang.org/x/tools/go/ast/astutil](https://golang.org/x/tools/go/ast/astutil), and provides the `Apply` function with similar semantics.

## Imports

The decorator can automatically manage the `import` block, which is a non-trivial task.

Use `NewDecoratorWithImports` and `NewRestorerWithImports` to create an import aware decorator / restorer.

During decoration, remote identifiers are normalised - `*ast.SelectorExpr` nodes that represent qualified identifiers are replaced with `*dst.Ident` nodes with the `Path` field set to the path of the imported package.

When adding a qualified identifier node, there is no need to use `*dst.SelectorExpr` - just add a `*dst.Ident` and set `Path` to the imported package path. The restorer will wrap it in a `*ast.SelectorExpr` where appropriate when converting back to ast, and also update the import block.

To enable import management, the decorator must be able to resolve the imported package for selector expressions and identifiers, and the restorer must be able to resolve the name of a package given its path. Several implementations for these resolvers are provided, and the best method will depend on the environment. See below for more details.

## Load

The `Load` convenience function uses `go/packages` to load packages and decorate all loaded ast files, with import management enabled:

```
1 // Create a simple module in a temporary directory
2 dir, err := tempDir(map[string]string{
3     "go.mod": "module root",
4     "main.go": "package main \n\n func main() {}"},
5 })
6 defer os.RemoveAll(dir)
7 if err != nil {
8     panic(err)
9 }
10
11 // Use the Load convenience function that calls go/packages to load the
12 // package. All loaded
13 // ast files are decorated to dst.
```



---

```

13 pkgs, err := decorator.Load(&packages.Config{Dir: dir, Mode: packages.
    LoadSyntax}, "root")
14 if err != nil {
15     panic(err)
16 }
17 p := pkgs[0]
18 f := p.Syntax[0]
19
20 // Add a call expression. Note we don't have to use a SelectorExpr -
    just adding an Ident with
21 // the imported package path will do. The restorer will add
    SelectorExpr where appropriate when
22 // converting back to ast. Note the new Path field on *dst.Ident. Set
    this to the package path
23 // of the imported package, and the restorer will automatically add the
    import to the import
24 // block.
25 b := f.Decls[0].(*dst.FuncDecl).Body
26 b.List = append(b.List, &dst.ExprStmt{
27     X: &dst.CallExpr{
28         Fun: &dst.Ident{Path: "fmt", Name: "Println"},
29         Args: []dst.Expr{
30             &dst.BasicLit{Kind: token.STRING, Value: strconv.Quote("
                Hello, World!")},
31         },
32     },
33 })
34
35 // Create a restorer with the import manager enabled, and print the
    result. As you can see, the
36 // import block is automatically managed, and the Println ident is
    converted to a SelectorExpr:
37 r := decorator.NewRestorerWithImports("root", gopackages.New(dir))
38 if err := r.Print(p.Syntax[0]); err != nil {
39     panic(err)
40 }
41
42 //Output:
43 //package main
44 //
45 //import "fmt"
46 //
47 //func main() { fmt.Println("Hello, World!") }

```

## Mappings

The decorator exposes `Dst.Nodes` and `Ast.Nodes` which map between `ast.Node` and `dst.Node`. This enables systems that refer to `ast` nodes (such as `go/types`) to be used:

---

```

1 code := `package main
2
3 func main() {
4     var i int
5     i++
6     println(i)
7 }`
8
9 // Parse the code to AST
10 fset := token.NewFileSet()
11 astFile, err := parser.ParseFile(fset, "", code, parser.ParseComments)
12 if err != nil {
13     panic(err)
14 }
15
16 // Invoke the type checker using AST as input
17 typesInfo := types.Info{
18     Defs:  make(map[*ast.Ident]types.Object),
19     Uses:  make(map[*ast.Ident]types.Object),
20 }
21 conf := &types.Config{}
22 if _, err := conf.Check("", fset, []*ast.File{astFile}, &typesInfo);
23     err != nil {
24     panic(err)
25 }
26
27 // Create a new decorator, which will track the mapping between ast and
28     dst nodes
29 dec := decorator.NewDecorator(fset)
30
31 // Decorate the *ast.File to give us a *dst.File
32 f, err := dec.DecorateFile(astFile)
33 if err != nil {
34     panic(err)
35 }
36
37 // Find the *dst.Ident for the definition of "i"
38 dstDef := f.Decls[0].(*dst.FuncDecl).Body.List[0].(*dst.DeclStmt).Decl
39     .(*dst.GenDecl).Specs[0].(*dst.ValueSpec).Names[0]
40
41 // Find the *ast.Ident using the Ast.Nodes mapping
42 astDef := dec.Ast.Nodes[dstDef].(*ast.Ident)
43
44 // Find the types.Object corresponding to "i"
45 obj := typesInfo.Defs[astDef]
46
47 // Find all the uses of that object
48 var astUses []*ast.Ident
49 for id, ob := range typesInfo.Uses {
50     if ob != obj {

```

---

```

48         continue
49     }
50     astUses = append(astUses, id)
51 }
52
53 // Find each *dst.Ident in the Dst.Nodes mapping
54 var dstUses []*dst.Ident
55 for _, id := range astUses {
56     dstUses = append(dstUses, dec.Dst.Nodes[id].(*dst.Ident))
57 }
58
59 // Change the name of the original definition and all uses
60 dstDef.Name = "foo"
61 for _, id := range dstUses {
62     id.Name = "foo"
63 }
64
65 // Print the DST
66 if err := decorator.Print(f); err != nil {
67     panic(err)
68 }
69
70 //Output:
71 //package main
72 //
73 //func main() {
74 //    var foo int
75 //    foo++
76 //    println(foo)
77 //}

```

## Resolvers

There are two separate interfaces defined by the resolver package which allow the decorator and restorer to automatically manage the imports block.

The decorator uses a [DecoratorResolver](#) which resolves the package path of any `*ast.Ident`. This is complicated by dot-import syntax (see below).

The restorer uses a [RestorerResolver](#) which resolves the name of any package given the path. This is complicated by vendoring and Go modules.

When [Resolver](#) is set on [Decorator](#) or [Restorer](#), the [Path](#) property must be set to the local package path.

Several implementations of both interfaces that are suitable for different environments are provided:

---

## DecoratorResolver

**gotypes** The gotypes package provides a [DecoratorResolver](#) with full dot-import compatibility. However it requires full export data for all imported packages, so the [Uses](#) map from [go/types.Info](#) is required. There are several methods of generating [go/types.Info](#). Using [golang.org/x/tools/go/packages.Load](#) is recommended for full Go modules compatibility. See the [decorator.Load](#) convenience function to automate this.

**goast** The goast package provides a simplified [DecoratorResolver](#) that only needs to scan a single ast file. This is unable to resolve identifiers from dot-imported packages, so will panic if a dot-import is encountered in the import block. It uses the provided [RestorerResolver](#) to resolve the names of all imported packages. If no [RestorerResolver](#) is provided, the guess implementation is used.

## RestorerResolver

**gopackages** The gopackages package provides a [RestorerResolver](#) with full compatibility with Go modules. It uses [golang.org/x/tools/go/packages](#) to load the package data. This may be very slow, and uses the [go](#) command line tool to query package data, so may not be compatible with some environments.

**gobuild** The gobuild package provides an alternative [RestorerResolver](#) that uses the legacy [go/build](#) system to load the imported package data. This may be needed in some circumstances and provides better performance than [go/packages](#). However, this is not Go modules aware.

**guess and simple** The guess and simple packages provide simple [RestorerResolver](#) implementations that may be useful in certain circumstances, or where performance is critical. [simple](#) resolves paths only if they occur in a provided map. [guess](#) guesses the package name based on the last part of the path.

## Example

Here's an example of supplying resolvers for the decorator and restorer:

```
1 code := `package main
2
3     import "fmt"
4
```

---

```

5     func main() {
6         fmt.Println("a")
7     }`
8
9     dec := decorator.NewDecoratorWithImports(token.NewFileSet(), "main",
10    goast.New())
11
12     f, err := dec.Parse(code)
13     if err != nil {
14         panic(err)
15     }
16
17     f.Decls[1].(*dst.FuncDecl).Body.List[0].(*dst.ExprStmt).X.(*dst.
18     CallExpr).Args = []dst.Expr{
19         &dst.CallExpr{
20             Fun: &dst.Ident{Name: "A", Path: "foo.bar/baz"},
21         },
22     }
23
24     res := decorator.NewRestorerWithImports("main", guess.New())
25     if err := res.Print(f); err != nil {
26         panic(err)
27     }
28
29     //Output:
30     //package main
31     //
32     //import (
33     //    "fmt"
34     //    "foo.bar/baz"
35     //)
36     //func main() {
37     //    fmt.Println(baz.A())
38     //}

```

## Alias

To control the alias of imports, use a `FileRestorer`:

```

1     code := `package main
2
3     import "fmt"
4
5     func main() {
6         fmt.Println("a")
7     }`
8

```

---

```
9  dec := decorator.NewDecoratorWithImports(token.NewFileSet(), "main",
    goast.New())
10
11  f, err := dec.Parse(code)
12  if err != nil {
13      panic(err)
14  }
15
16  res := decorator.NewRestorerWithImports("main", guess.New())
17
18  fr := res.FileRestorer()
19  fr.Alias["fmt"] = "fmt1"
20
21  if err := fr.Print(f); err != nil {
22      panic(err)
23  }
24
25  //Output:
26  //package main
27  //
28  //import fmt1 "fmt"
29  //
30  //func main() {
31  //    fmt1.Println("a")
32  //}
```

## Details

For more information on exactly how the imports block is managed, read through the test cases.

## Dot-imports

Consider this file...

```
1  package main
2
3  import (
4      . "a"
5  )
6
7  func main() {
8      B()
9      C()
10 }
```

`B` and `C` could be local identifiers from a different file in this package, or from the imported package `a`. If only one is from `a` and it is removed, we should remove the import when we restore to `ast`. Thus

---

the resolver needs to be able to resolve the package using the full info from [go/types](#).

## **Status**

This package is well tested and used in many projects. The API should be considered stable going forward.

## **Chat?**

Feel free to create an issue or chat in the [#dst](#) Gophers Slack channel.

## **Contributing**

For further developing or contributing to [dst](#), check out these notes.

## **Special thanks**

Thanks very much to [hawkinsw](#) for taking on the task of adding generics compatibility to [dst](#).