



# ***UNDERSTANDING THE GO COMPILER***

JESÚS ESPINO - SOFTWARE ENGINEER @ MATTERMOST





# INTRODUCTION



DISCLAIMER



# OUR EXAMPLE CODE



```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



# THE GO COMPILER

(1.19)

# THE GO COMPILER

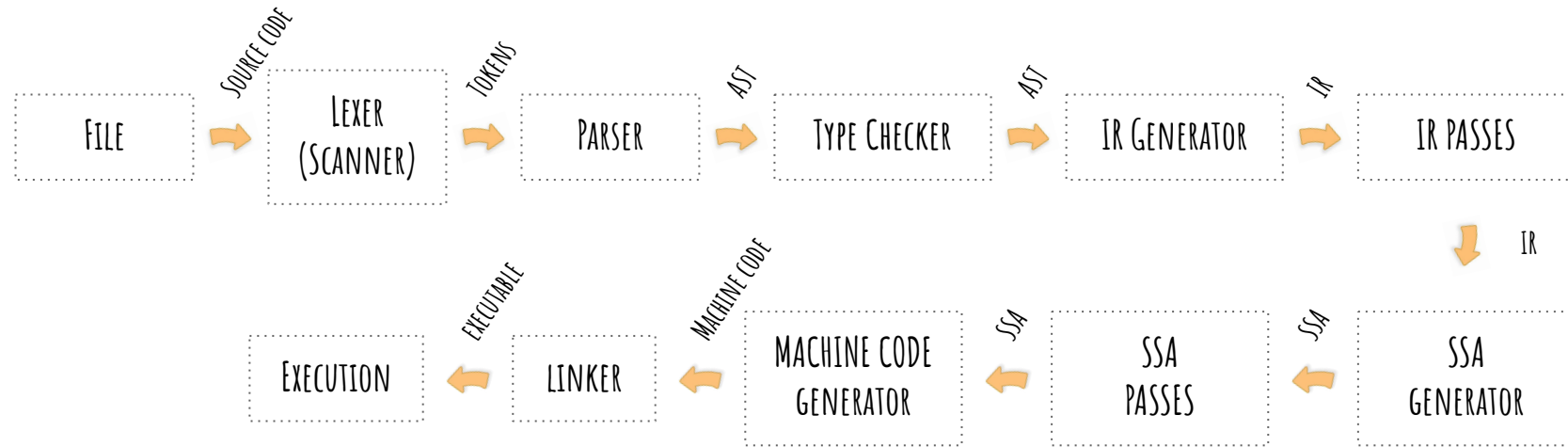




# THE GO COMPILER

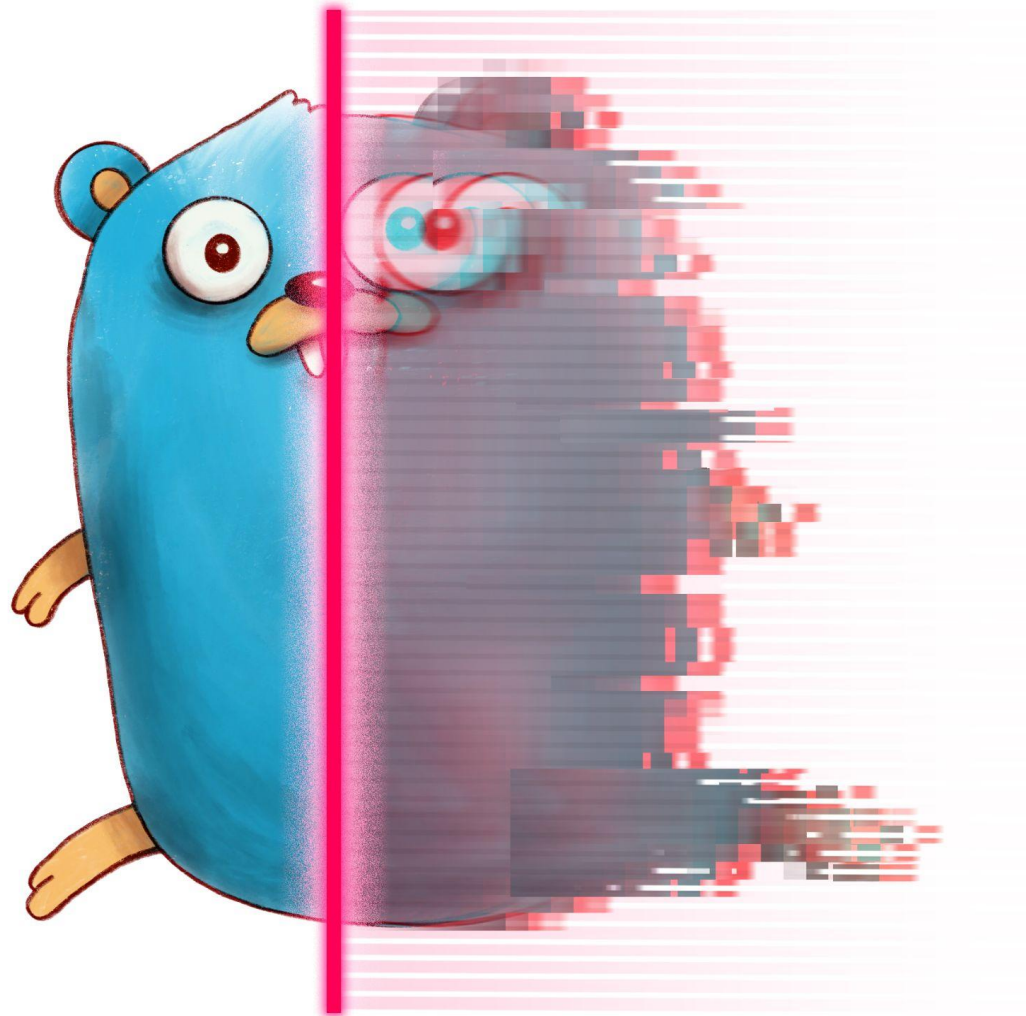


# PHASES OF THE GO COMPILER

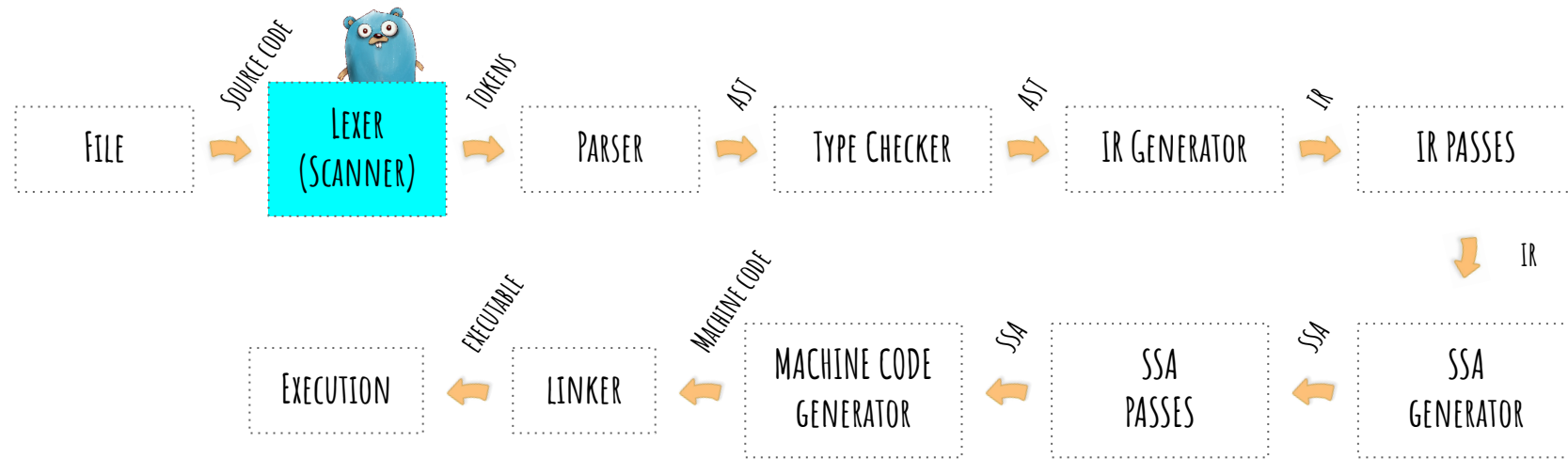




# THE LEXER (SCANNER)



# THE LEXER (SCANNER)





# THE LEXER (SCANNER)

```
package main
```



```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```

# THE LEXER (SCANNER)

```
package main
```



```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



# THE LEXER (SCANNER)

```
package main
```



```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```

# THE LEXER (SCANNER)

```
package main
```



```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



# THE LEXER (SCANNER)

```
package main
```



```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```

# THE LEXER (SCANNER)

```
package main
```



```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```

# THE LEXER (SCANNER)

```
package main
```



```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



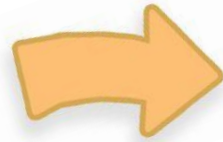
# THE LEXER (SCANNER)

```
package main
```



```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



Line 1: package (package)

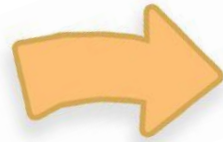
# THE LEXER (SCANNER)

```
package main
```



```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



```
Line 1: package (package)
```

```
Line 1: IDENT (main)
```

```
Line 1: ; ()
```

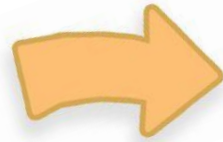
# THE LEXER (SCANNER)

```
package main
```

```
import "fmt"
```



```
func main() {  
    fmt.Println("hello world!")  
}
```



```
Line 1: package (package)
```

```
Line 1: IDENT (main)
```

```
Line 1: ; ()
```

```
Line 3: import (import)
```

```
Line 3: STRING ("fmt")
```

```
Line 3: ; ()
```

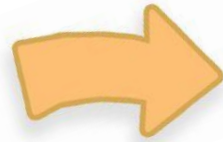


# THE LEXER (SCANNER)

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



```
Line 1: package (package)
```

```
Line 1: IDENT (main)
```

```
Line 1: ; ()
```

```
Line 3: import (import)
```

```
Line 3: STRING ("fmt")
```

```
Line 3: ; ()
```

```
Line 5: func (func)
```

```
Line 5: IDENT (main)
```

```
Line 5: ( ()
```

```
Line 5: ) ()
```

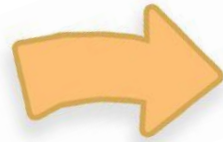
```
Line 5: { ()
```

# THE LEXER (SCANNER)

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



```
Line 1: package (package)  
Line 1: IDENT (main)  
Line 1: ; ()  
Line 3: import (import)  
Line 3: STRING ("fmt")  
Line 3: ; ()  
Line 5: func (func)  
Line 5: IDENT (main)  
Line 5: ( ()  
Line 5: ) ()  
Line 5: { ()  
Line 6: IDENT (fmt)  
Line 6: . ()  
Line 6: IDENT (Println)  
Line 6: ( ()  
Line 6: STRING ("hello-world!")  
Line 6: ) ()  
Line 6: ; ()  
Line 7: } ()  
Line 7: ; ()
```

# SAMPLE SCANNER CASES

```
case ';':  
    tok = token.SEMICOLON  
    lit = ";
```

```
case '.':  
    tok = token.PERIOD  
    if s.ch == '.' && s.peek() == '.' {  
        s.next()  
        s.next() // consume last '.'  
        tok = token.ELLIPSIS  
    }
```

```
func (s *Scanner) switch2(tok0, tok1 token.Token) token.Token {  
    if s.ch == '=' {  
        s.next()  
        return tok1  
    }  
    return tok0  
}  
...  
case '*':  
    tok = s.switch2(token.MUL, token.MUL_ASSIGN)
```



# SAMPLE SCANNER CASES

```
case ';;':
```

```
    tok = token.SEMICOLON
```

```
    lit = ";;"
```

```
case '...':
```

```
    tok = token.PERIOD
```

```
    if s.ch == '.' && s.peek() == '.' {
```

```
        s.next()
```

```
        s.next() // consume last '.'
```

```
        tok = token.ELLIPSIS
```

```
    }
```

```
func (s *Scanner) switch2(tok0, tok1 token.Token) token.Token {
```

```
    if s.ch == '=' {
```

```
        s.next()
```

```
        return tok1
```

```
    }
```

```
    return tok0
```

```
}
```

```
...
```

```
case '*':
```

```
    tok = s.switch2(token.MUL, token.MUL_ASSIGN)
```

# SAMPLE SCANNER CASES

```
case ';;':  
    tok = token.SEMICOLON  
    lit = ";;"
```

```
case '...':  
    tok = token.PERIOD  
    if s.ch == '.' && s.peek() == '.' {  
        s.next()  
        s.next() // consume last '.'  
        tok = token.ELLIPSIS  
    }
```

```
func (s *Scanner) switch2(tok0, tok1 token.Token) token.Token {  
    if s.ch == '=' {  
        s.next()  
        return tok1  
    }  
    return tok0  
}  
...  
case '*':  
    tok = s.switch2(token.MUL, token.MUL_ASSIGN)
```

# SAMPLE SCANNER CASES

```
case ';':  
    tok = token.SEMICOLON  
    lit = ";
```

```
case '.':  
    tok = token.PERIOD  
    if s.ch == '.' && s.peek() == '.' {  
        s.next()  
        s.next() // consume last '.'  
        tok = token.ELLIPSIS  
    }
```

```
func (s *Scanner) switch2(tok0, tok1 token.Token) token.Token {  
    if s.ch == '=' {  
        s.next()  
        return tok1  
    }  
    return tok0  
}  
...  
case '*':  
    tok = s.switch2(token.MUL, token.MUL_ASSIGN)
```

# SAMPLE SCANNER CASES

```
case ';':  
    tok = token.SEMICOLON  
    lit = ";
```

```
case '.':  
    tok = token.PERIOD  
    if s.ch == '.' && s.peek() == '.' {  
        s.next()  
        s.next() // consume last '.'  
        tok = token.ELLIPSIS  
    }
```

```
func (s *Scanner) switch2(tok0, tok1 token.Token) token.Token {  
    if s.ch == '=' {  
        s.next()  
        return tok1  
    }  
    return tok0  
}  
...  
case '*':  
    tok = s.switch2(token.MUL, token.MUL_ASSIGN)
```



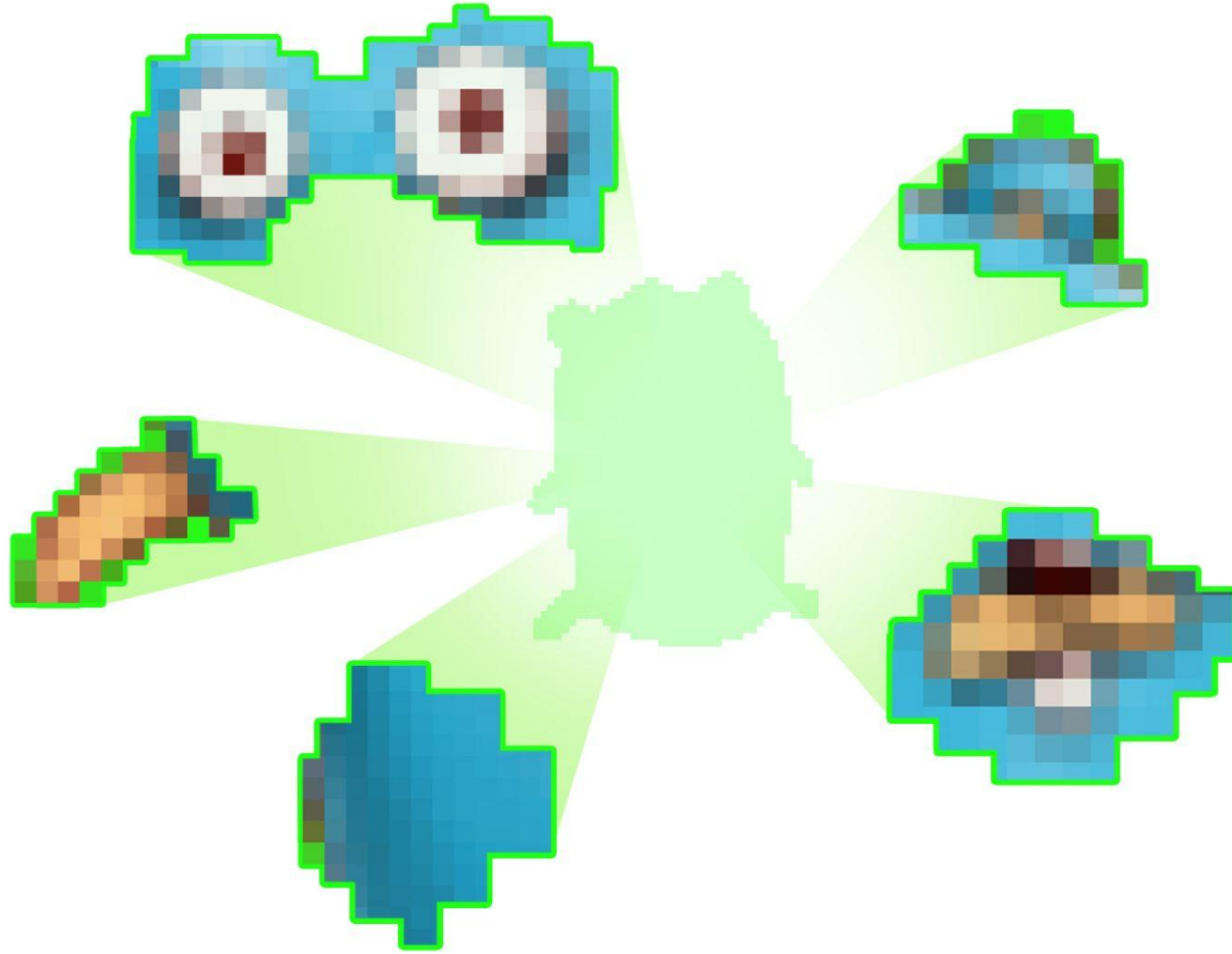
# SAMPLE SCANNER CASES

```
case ';':  
    tok = token.SEMICOLON  
    lit = ";
```

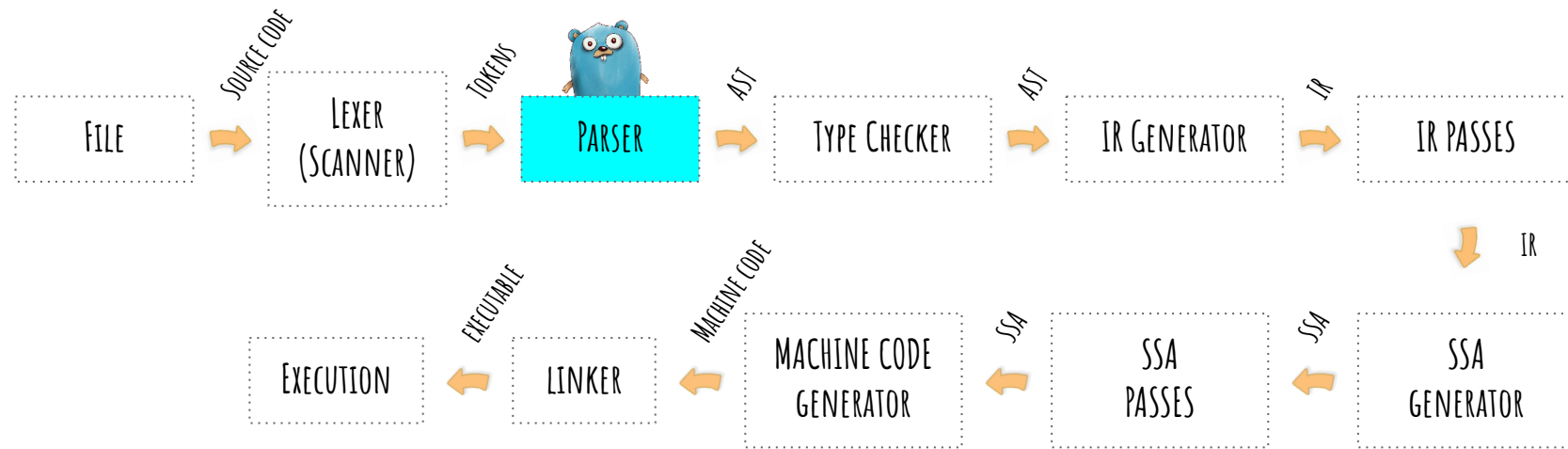
```
case '.':  
    tok = token.PERIOD  
    if s.ch == '.' && s.peek() == '.' {  
        s.next()  
        s.next() // consume last '.'  
        tok = token.ELLIPSIS  
    }
```

```
func (s *Scanner) switch2(tok0, tok1 token.Token) token.Token {  
    if s.ch == '=' {  
        s.next()  
        return tok1  
    }  
    return tok0  
}  
...  
case '*':  
    tok = s.switch2(token.MUL, token.MUL_ASSIGN)
```

# THE PARSER



# THE PARSER

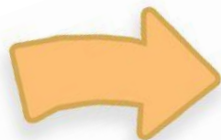


# THE PARSER

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



```
0  *ast.File {  
1  .   Package: helloworld.go:1:1  
2  .   Name: *ast.Ident {  
3  .     .   NamePos: helloworld.go:1:9  
4  .     .   Name: "main"  
5  .     }  
6  .   Decls: []ast.Decl (len = 2) {  
7  .     .   0: *ast.GenDecl {  
8  .     .     .   TokPos: helloworld.go:3:1  
9  .     .     .   Tok: import  
10 .     .     .   Lparen: -  
11 .     .     .   Specs: []ast.Spec (len = 1) {  
12 .     .     .     .   0: *ast.ImportSpec {  
13 .     .     .     .     .   Path: *ast.BasicLit {  
14 .     .     .     .     .     .   ValuePos: helloworld.go:3:8  
15 .     .     .     .     .     .   Kind: STRING  
16 .     .     .     .     .     .   Value: "\"fmt\""  
17 .     .     .     .     .     }  
18 .     .     .     .     .   EndPos: -  
19 .     .     .     .     }  
20 .     .     .     }  
21 .     .     .   Rparen: -  
22 .     .     }  
23 .     .   1: *ast.FuncDecl {  
24 .     .     .   Name: *ast.Ident {  
25 .     .     .     .   NamePos: helloworld.go:5:6  
26 .     .     .     .   Name: "main"  
27 .     .     .     .   Obj: *ast.Object {  
28 .     .     .     .     .   Kind: func  
29 .     .     .     .     .   Name: "main"  
30 .     .     .     .     .   Decl: *(obj @ 23)  
31 .     .     .     .     }  
32 .     .     .     }  
33 .     .     .   Type: *ast.FuncType {  
34 .     .     .     .   Func: helloworld.go:5:1  
35 .     .     .     .   Params: *ast.FieldList {  
36 .     .     .     .     .   Opening: helloworld.go:5:10  
37 .     .     .     .     .   Closing: helloworld.go:5:11  
38 .     .     .     .     }  
39 .     .     .     }  
40 .     .     }  
41 .     }  
42 .   }
```

```
40 . . .   Body: *ast.BlockStmt {  
41 . . .     .   Lbrace: helloworld.go:5:13  
42 . . .     .   List: []ast.Stmt (len = 1) {  
43 . . .     .     .   0: *ast.ExprStmt {  
44 . . .     .     .     .   X: *ast.CallExpr {  
45 . . .     .     .     .     .   Fun: *ast.SelectorExpr {  
46 . . .     .     .     .     .     .   X: *ast.Ident {  
47 . . .     .     .     .     .     .     .   NamePos: helloworld.go:6:2  
48 . . .     .     .     .     .     .     .   Name: "fmt"  
49 . . .     .     .     .     .     .     }  
50 . . .     .     .     .     .     .   Sel: *ast.Ident {  
51 . . .     .     .     .     .     .     .     .   NamePos: helloworld.go:6:6  
52 . . .     .     .     .     .     .     .     .   Name: "Println"  
53 . . .     .     .     .     .     .     .     }  
54 . . .     .     .     .     .     .     }  
55 . . .     .     .     .     .     .   Lparen: helloworld.go:6:13  
56 . . .     .     .     .     .     .   Args: []ast.Expr (len = 1) {  
57 . . .     .     .     .     .     .     .   0: *ast.BasicLit {  
58 . . .     .     .     .     .     .     .     .   ValuePos: helloworld.go:6:14  
59 . . .     .     .     .     .     .     .     .   Kind: STRING  
60 . . .     .     .     .     .     .     .     .   Value: "\"hello-world!\""  
61 . . .     .     .     .     .     .     .     }  
62 . . .     .     .     .     .     .     }  
63 . . .     .     .     .     .     .   Ellipsis: -  
64 . . .     .     .     .     .     .   Rparen: helloworld.go:6:28  
65 . . .     .     .     .     .     }  
66 . . .     .     .     .     }  
67 . . .     .     .     }  
68 . . .     .     .   Rbrace: helloworld.go:7:1  
69 . . .     .     }  
70 . . .     }  
71 . . .   }  
72 .   Scope: *ast.Scope {  
73 .     .   Objects: map[string]*ast.Object (len = 1) {  
74 .     .     .   "main": *(obj @ 27)  
75 .     .     }  
76 .     .   }  
77 .   Imports: []*ast.ImportSpec (len = 1) {  
78 .     .   0: *(obj @ 12)  
79 .     }  
80 .   Unresolved: []*ast.Ident (len = 1) {  
81 .     .   0: *(obj @ 46)  
82 .     }  
83 . }
```

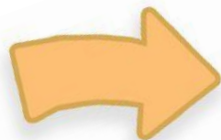


# THE PARSER

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



```
0  *ast.File {  
1  .   Package: helloworld.go:1:1  
2  .   Name: *ast.Ident {  
3  .       NamePos: helloworld.go:1:9  
4  .       Name: "main"  
5  .   }  
6  .   Decls: []ast.Decl (len = 2) {  
7  .       0: *ast.GenDecl {  
8  .           TokPos: helloworld.go:3:1  
9  .           Tok: import  
10 .           Lparen: -  
11 .           Specs: []ast.Spec (len = 1) {  
12 .               0: *ast.ImportSpec {  
13 .                   Path: *ast.BasicLit {  
14 .                       ValuePos: helloworld.go:3:8  
15 .                       Kind: STRING  
16 .                       Value: "\"fmt\""  
17 .                   }  
18 .                   EndPos: -  
19 .               }  
20 .           }  
21 .           Rparen: -  
22 .       }  
23 .       1: *ast.FuncDecl {  
24 .           Name: *ast.Ident {  
25 .               NamePos: helloworld.go:5:6  
26 .               Name: "main"  
27 .               Obj: *ast.Object {  
28 .                   Kind: func  
29 .                   Name: "main"  
30 .                   Decl: *(obj @ 23)  
31 .               }  
32 .           }  
33 .           Type: *ast.FuncType {  
34 .               Func: helloworld.go:5:1  
35 .               Params: *ast.FieldList {  
36 .                   Opening: helloworld.go:5:10  
37 .                   Closing: helloworld.go:5:11  
38 .               }  
39 .           }  
40 .       }  
41 .   }  
42 . }
```

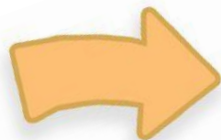
```
40 .   .   Body: *ast.BlockStmt {  
41 .       .   Lbrace: helloworld.go:5:13  
42 .       .   List: []ast.Stmt (len = 1) {  
43 .           .   0: *ast.ExprStmt {  
44 .               .   X: *ast.CallExpr {  
45 .                   .   Fun: *ast.SelectorExpr {  
46 .                       .   X: *ast.Ident {  
47 .                           NamePos: helloworld.go:6:2  
48 .                           Name: "fmt"  
49 .                       }  
50 .                       Sel: *ast.Ident {  
51 .                           NamePos: helloworld.go:6:6  
52 .                           Name: "Println"  
53 .                       }  
54 .                   }  
55 .                   Lparen: helloworld.go:6:13  
56 .                   Args: []ast.Expr (len = 1) {  
57 .                       .   0: *ast.BasicLit {  
58 .                           ValuePos: helloworld.go:6:14  
59 .                           Kind: STRING  
60 .                           Value: "\"hello-world!\""  
61 .                       }  
62 .                   }  
63 .                   Ellipsis: -  
64 .                   Rparen: helloworld.go:6:28  
65 .               }  
66 .           }  
67 .       }  
68 .       Rbrace: helloworld.go:7:1  
69 .   }  
70 . }  
71 . }  
72 . Scope: *ast.Scope {  
73 .     .   Objects: map[string]*ast.Object (len = 1) {  
74 .         .   "main": *(obj @ 27)  
75 .     .   }  
76 .     }  
77 .     Imports: []*ast.ImportSpec (len = 1) {  
78 .         .   0: *(obj @ 12)  
79 .     .   }  
80 .     Unresolved: []*ast.Ident (len = 1) {  
81 .         .   0: *(obj @ 46)  
82 .     .   }  
83 . }
```

# THE PARSER

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



```
0  *ast.File {  
1  .   Package: helloworld.go:1:1  
2  .   Name: *ast.Ident {  
3  . .   NamePos: helloworld.go:1:9  
4  . .   Name: "main"  
5  .   }  
6  .   Decls: []ast.Decl (len = 2) {  
7  . .   0: *ast.GenDecl {  
8  . . .   TokPos: helloworld.go:3:1  
9  . . .   Tok: import  
10 . . .   Lparen: -  
11 . . .   Specs: []ast.Spec (len = 1) {  
12 . . . .   0: *ast.ImportSpec {  
13 . . . . .   Path: *ast.BasicLit {  
14 . . . . . .   ValuePos: helloworld.go:3:8  
15 . . . . . .   Kind: STRING  
16 . . . . . .   Value: "\"fmt\""  
17 . . . . .   }  
18 . . . . .   EndPos: -  
19 . . . .   }  
20 . . .   }  
21 . .   Rparen: -  
22 .   }  
23 . .   1: *ast.FuncDecl {  
24 . . .   Name: *ast.Ident {  
25 . . . .   NamePos: helloworld.go:5:6  
26 . . . .   Name: "main"  
27 . . . .   Obj: *ast.Object {  
28 . . . . .   Kind: func  
29 . . . . .   Name: "main"  
30 . . . . .   Decl: *(obj @ 23)  
31 . . . .   }  
32 . . .   }  
33 . . .   Type: *ast.FuncType {  
34 . . . .   Func: helloworld.go:5:1  
35 . . . .   Params: *ast.FieldList {  
36 . . . . .   Opening: helloworld.go:5:10  
37 . . . . .   Closing: helloworld.go:5:11  
38 . . . .   }  
39 . . .   }
```

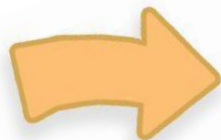
```
40 . . .   Body: *ast.BlockStmt {  
41 . . . .   Lbrace: helloworld.go:5:13  
42 . . . .   List: []ast.Stmt (len = 1) {  
43 . . . . .   0: *ast.ExprStmt {  
44 . . . . . .   X: *ast.CallExpr {  
45 . . . . . . .   Fun: *ast.SelectorExpr {  
46 . . . . . . . .   X: *ast.Ident {  
47 . . . . . . . . .   NamePos: helloworld.go:6:2  
48 . . . . . . . . .   Name: "fmt"  
49 . . . . . . . .   }  
50 . . . . . . . .   Sel: *ast.Ident {  
51 . . . . . . . . .   NamePos: helloworld.go:6:6  
52 . . . . . . . . .   Name: "Println"  
53 . . . . . . . .   }  
54 . . . . . . .   }  
55 . . . . . .   Lparen: helloworld.go:6:13  
56 . . . . . .   Args: []ast.Expr (len = 1) {  
57 . . . . . . .   0: *ast.BasicLit {  
58 . . . . . . . .   ValuePos: helloworld.go:6:14  
59 . . . . . . . .   Kind: STRING  
60 . . . . . . . .   Value: "\"hello-world!\""  
61 . . . . . . . .   }  
62 . . . . . . .   }  
63 . . . . . .   Ellipsis: -  
64 . . . . . .   Rparen: helloworld.go:6:28  
65 . . . . .   }  
66 . . . .   }  
67 . . .   }  
68 . . .   Rbrace: helloworld.go:7:1  
69 . .   }  
70 .   }  
71 .   }  
72 .   Scope: *ast.Scope {  
73 . .   Objects: map[string]*ast.Object (len = 1) {  
74 . . .   "main": *(obj @ 27)  
75 . .   }  
76 .   }  
77 .   Imports: []*ast.ImportSpec (len = 1) {  
78 . .   0: *(obj @ 12)  
79 .   }  
80 .   Unresolved: []*ast.Ident (len = 1) {  
81 . .   0: *(obj @ 46)  
82 .   }  
83 }
```

# THE PARSER

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



```
0  *ast.File {  
1  .   Package: helloworld.go:1:1  
2  .   Name: *ast.Ident {  
3  .     .   NamePos: helloworld.go:1:9  
4  .     .   Name: "main"  
5  .     }  
6  .   Decls: []ast.Decl (len = 2) {  
7  .     .   0: *ast.GenDecl {  
8  .     .     .   TokPos: helloworld.go:3:1  
9  .     .     .   Tok: import  
10 .     .     .   Lparen: -  
11 .     .     .   Specs: []ast.Spec (len = 1) {  
12 .     .     .     .   0: *ast.ImportSpec {  
13 .     .     .     .     .   Path: *ast.BasicLit {  
14 .     .     .     .     .     .   ValuePos: helloworld.go:3:8  
15 .     .     .     .     .     .   Kind: STRING  
16 .     .     .     .     .     .   Value: "\"fmt\""  
17 .     .     .     .     .     }  
18 .     .     .     .     .   EndPos: -  
19 .     .     .     .     }  
20 .     .     .     }  
21 .     .     .   Rparen: -  
22 .     .     }  
23 .     .   1: *ast.FuncDecl {  
24 .     .     .   Name: *ast.Ident {  
25 .     .     .     .   NamePos: helloworld.go:5:6  
26 .     .     .     .   Name: "main"  
27 .     .     .     .   Obj: *ast.Object {  
28 .     .     .     .     .   Kind: func  
29 .     .     .     .     .   Name: "main"  
30 .     .     .     .     .   Decl: *(obj @ 23)  
31 .     .     .     .     }  
32 .     .     .     .     }  
33 .     .     .     .   Type: *ast.FuncType {  
34 .     .     .     .     .   Func: helloworld.go:5:1  
35 .     .     .     .     .   Params: *ast.FieldList {  
36 .     .     .     .     .     .   Opening: helloworld.go:5:10  
37 .     .     .     .     .     .   Closing: helloworld.go:5:11  
38 .     .     .     .     .     }  
39 .     .     .     .     }  
40 .     .     .     }  
41 .     .     }  
42 .     }  
43 .   }
```

```
40 . . .   Body: *ast.BlockStmt {  
41 . . .     .   Lbrace: helloworld.go:5:13  
42 . . .     .   List: []ast.Stmt (len = 1) {  
43 . . .     .     .   0: *ast.ExprStmt {  
44 . . .     .     .     .   X: *ast.CallExpr {  
45 . . .     .     .     .     .   Fun: *ast.SelectorExpr {  
46 . . .     .     .     .     .     .   X: *ast.Ident {  
47 . . .     .     .     .     .     .     .   NamePos: helloworld.go:6:2  
48 . . .     .     .     .     .     .     .   Name: "fmt"  
49 . . .     .     .     .     .     .     }  
50 . . .     .     .     .     .     .   Sel: *ast.Ident {  
51 . . .     .     .     .     .     .     .     .   NamePos: helloworld.go:6:6  
52 . . .     .     .     .     .     .     .     .   Name: "Println"  
53 . . .     .     .     .     .     .     .     }  
54 . . .     .     .     .     .     .     }  
55 . . .     .     .     .     .     .   Lparen: helloworld.go:6:13  
56 . . .     .     .     .     .     .   Args: []ast.Expr (len = 1) {  
57 . . .     .     .     .     .     .     .   0: *ast.BasicLit {  
58 . . .     .     .     .     .     .     .     .   ValuePos: helloworld.go:6:14  
59 . . .     .     .     .     .     .     .     .   Kind: STRING  
60 . . .     .     .     .     .     .     .     .   Value: "\"hello-world!\""  
61 . . .     .     .     .     .     .     .     }  
62 . . .     .     .     .     .     .     }  
63 . . .     .     .     .     .     .   Ellipsis: -  
64 . . .     .     .     .     .     .   Rparen: helloworld.go:6:28  
65 . . .     .     .     .     .     }  
66 . . .     .     .     .     }  
67 . . .     .     .     }  
68 . . .     .   Rbrace: helloworld.go:7:1  
69 . . .     }  
70 . . .   }  
71 . . }  
72 .   Scope: *ast.Scope {  
73 .     .   Objects: map[string]*ast.Object (len = 1) {  
74 .     .     .   "main": *(obj @ 27)  
75 .     .     }  
76 .     .   }  
77 .   Imports: []*ast.ImportSpec (len = 1) {  
78 .     .   0: *(obj @ 12)  
79 .     }  
80 .   Unresolved: []*ast.Ident (len = 1) {  
81 .     .   0: *(obj @ 46)  
82 .     }  
83 . }
```

# THE PARSER

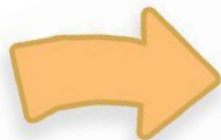
```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    fmt.Println("hello world!")
```

```
}
```



```
0  *ast.File {
1  .   Package: helloworld.go:1:1
2  .   Name: *ast.Ident {
3  .     .   NamePos: helloworld.go:1:9
4  .     .   Name: "main"
5  .     }
6  .   Decls: []ast.Decl (len = 2) {
7  .     .   0: *ast.GenDecl {
8  .     .     .   TokPos: helloworld.go:3:1
9  .     .     .   Tok: import
10 .     .     .   Lparen: -
11 .     .     .   Specs: []ast.Spec (len = 1) {
12 .     .     .     .   0: *ast.ImportSpec {
13 .     .     .     .     .   Path: *ast.BasicLit {
14 .     .     .     .     .     .   ValuePos: helloworld.go:3:8
15 .     .     .     .     .     .   Kind: STRING
16 .     .     .     .     .     .   Value: "\"fmt\""
17 .     .     .     .     .     }
18 .     .     .     .     .   EndPos: -
19 .     .     .     .     }
20 .     .     .     }
21 .     .     .   Rparen: -
22 .     .     }
23 .     .   1: *ast.FuncDecl {
24 .     .     .   Name: *ast.Ident {
25 .     .     .     .   NamePos: helloworld.go:5:6
26 .     .     .     .   Name: "main"
27 .     .     .     .   Obj: *ast.Object {
28 .     .     .     .     .   Kind: func
29 .     .     .     .     .   Name: "main"
30 .     .     .     .     .   Decl: *(obj @ 23)
31 .     .     .     .     }
32 .     .     .     .   }
33 .     .     .   Type: *ast.FuncType {
34 .     .     .     .   Func: helloworld.go:5:1
35 .     .     .     .   Params: *ast.FieldList {
36 .     .     .     .     .   Opening: helloworld.go:5:10
37 .     .     .     .     .   Closing: helloworld.go:5:11
38 .     .     .     .     }
39 .     .     .     }

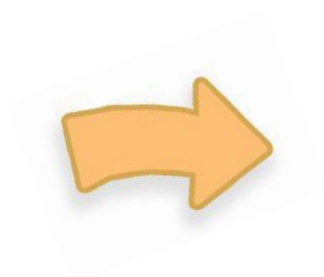
```

```
40 . . .   Body: *ast.BlockStmt {
41 . . .     .   Lbrace: helloworld.go:5:13
42 . . .     .   List: []ast.Stmt (len = 1) {
43 . . .     .     .   0: *ast.ExprStmt {
44 . . .     .     .     .   X: *ast.CallExpr {
45 . . .     .     .     .     .   Fun: *ast.SelectorExpr {
46 . . .     .     .     .     .     .   X: *ast.Ident {
47 . . .     .     .     .     .     .     .   NamePos: helloworld.go:6:2
48 . . .     .     .     .     .     .     .   Name: "fmt"
49 . . .     .     .     .     .     .     }
50 . . .     .     .     .     .     .   Sel: *ast.Ident {
51 . . .     .     .     .     .     .     .     .   NamePos: helloworld.go:6:6
52 . . .     .     .     .     .     .     .     .   Name: "Println"
53 . . .     .     .     .     .     .     .     }
54 . . .     .     .     .     .     .     }
55 . . .     .     .     .     .     .   Lparen: helloworld.go:6:13
56 . . .     .     .     .     .     .     .   Args: []ast.Expr (len = 1) {
57 . . .     .     .     .     .     .     .     .   0: *ast.BasicLit {
58 . . .     .     .     .     .     .     .     .     .   ValuePos: helloworld.go:6:14
59 . . .     .     .     .     .     .     .     .     .   Kind: STRING
60 . . .     .     .     .     .     .     .     .     .   Value: "\"hello-world!\""
61 . . .     .     .     .     .     .     .     .     }
62 . . .     .     .     .     .     .     .     }
63 . . .     .     .     .     .     .     .   Ellipsis: -
64 . . .     .     .     .     .     .     .   Rparen: helloworld.go:6:28
65 . . .     .     .     .     .     .     }
66 . . .     .     .     .     .     }
67 . . .     .     .     .     }
68 . . .     .   Rbrace: helloworld.go:7:1
69 . . .   }
70 .   }
71 . }
72 . Scope: *ast.Scope {
73 .   .   Objects: map[string]*ast.Object (len = 1) {
74 .   .     .   "main": *(obj @ 27)
75 .   .   }
76 .   }
77 .   Imports: []*ast.ImportSpec (len = 1) {
78 .   .   0: *(obj @ 12)
79 .   }
80 .   Unresolved: []*ast.Ident (len = 1) {
81 .   .   0: *(obj @ 46)
82 .   }
83 }
```

# THE PARSER



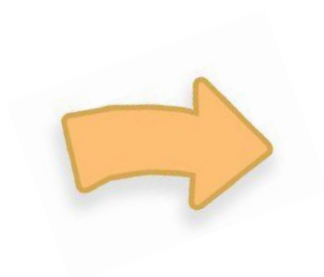
```
Line 1: package (package)
Line 1: IDENT (main)
Line 1: ; ()
Line 3: import (import)
Line 3: STRING ("fmt")
Line 3: ; ()
Line 5: func (func)
Line 5: IDENT (main)
Line 5: ( ()
Line 5: ) ()
Line 5: { ()
Line 6: IDENT (fmt)
Line 6: . ()
Line 6: IDENT (Println)
Line 6: ( ()
Line 6: STRING ("hello-world!")
Line 6: ) ()
Line 6: ; ()
Line 7: } ()
Line 7: ; ()
```



```
0 *ast.File {
1 . Package: helloworld.go:1:1
2 . Name: nil
6 . Decls: []ast.Decl (len = 0) {
71 . }
72 . Scope: *ast.Scope {
76 . }
77 . Imports: []*ast.ImportSpec (len = 0) {
79 . }
80 . Unresolved: []*ast.Ident (len = 0) {
82 . }
83 }
```

# THE PARSER

→ Line 1: package (package)  
Line 1: IDENT (main)  
Line 1: ; ()  
Line 3: import (import)  
Line 3: STRING ("fmt")  
Line 3: ; ()  
Line 5: func (func)  
Line 5: IDENT (main)  
Line 5: ( ()  
Line 5: ) ()  
Line 5: { ()  
Line 6: IDENT (fmt)  
Line 6: . ()  
Line 6: IDENT (Println)  
Line 6: ( ()  
Line 6: STRING ("hello-world!")  
Line 6: ) ()  
Line 6: ; ()  
Line 7: } ()  
Line 7: ; ()

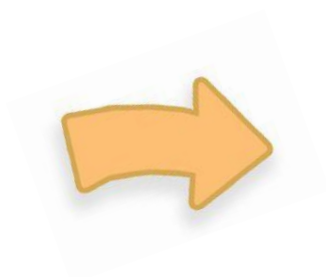


```
0 *ast.File {  
1 . Package: helloworld.go:1:1  
2 . Name: nil  
6 . Decls: []ast.Decl (len = 0) {  
71 . }  
72 . Scope: *ast.Scope {  
76 . }  
77 . Imports: []*ast.ImportSpec (len = 0) {  
79 . }  
80 . Unresolved: []*ast.Ident (len = 0) {  
82 . }  
83 }
```



# THE PARSER

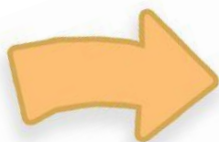
→ Line 1: package (package)  
Line 1: IDENT (main)  
Line 1: ; ()  
Line 3: import (import)  
Line 3: STRING ("fmt")  
Line 3: ; ()  
Line 5: func (func)  
Line 5: IDENT (main)  
Line 5: ( ()  
Line 5: ) ()  
Line 5: { ()  
Line 6: IDENT (fmt)  
Line 6: . ()  
Line 6: IDENT (Println)  
Line 6: ( ()  
Line 6: STRING ("hello-world!")  
Line 6: ) ()  
Line 6: ; ()  
Line 7: } ()  
Line 7: ; ()



```
0 *ast.File {  
1 . Package: helloworld.go:1:1  
2 . Name: *ast.Ident {  
3 . . NamePos: helloworld.go:1:9  
4 . . Name: "main"  
5 . }  
6 . Decls: []ast.Decl (len = 0) {  
71 . }  
72 . Scope: *ast.Scope {  
76 . }  
77 . Imports: []*ast.ImportSpec (len = 0) {  
79 . }  
80 . Unresolved: []*ast.Ident (len = 0) {  
82 . }  
83 }
```

# THE PARSER

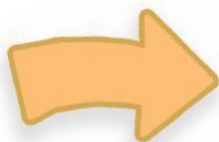
Line 1: package (package)  
Line 1: IDENT (main)  
Line 1: ; ()  
→ Line 3: import (import)  
Line 3: STRING ("fmt")  
Line 3: ; ()  
Line 5: func (func)  
Line 5: IDENT (main)  
Line 5: ( ()  
Line 5: ) ()  
Line 5: { ()  
Line 6: IDENT (fmt)  
Line 6: . ()  
Line 6: IDENT (Println)  
Line 6: ( ()  
Line 6: STRING ("hello-world!")  
Line 6: ) ()  
Line 6: ; ()  
Line 7: } ()  
Line 7: ; ()



```
0 *ast.File {
1 . Package: helloworld.go:1:1
2 . Name: *ast.Ident {
3 . . NamePos: helloworld.go:1:9
4 . . Name: "main"
5 . }
6 . Decls: []ast.Decl (len = 1) {
7 . . 0: *ast.GenDecl {
8 . . . TokPos: helloworld.go:3:1
9 . . . Tok: import
10 . . . Lparen: -
11 . . . Specs: []ast.Spec (len = 0) {
12 . . . . 0: *ast.ImportSpec {
20 . . . . }
21 . . . Rparen: -
22 . . }
71 . }
72 . Scope: *ast.Scope {
76 . }
77 . Imports: []*ast.ImportSpec (len = 0) {
79 . }
80 . Unresolved: []*ast.Ident (len = 0) {
82 . }
83 }
```

# THE PARSER

Line 1: package (package)  
Line 1: IDENT (main)  
Line 1: ; ()  
→ Line 3: import (import)  
Line 3: STRING ("fmt")  
Line 3: ; ()  
Line 5: func (func)  
Line 5: IDENT (main)  
Line 5: ( ()  
Line 5: ) ()  
Line 5: { ()  
Line 6: IDENT (fmt)  
Line 6: . ()  
Line 6: IDENT (Println)  
Line 6: ( ()  
Line 6: STRING ("hello-world!")  
Line 6: ) ()  
Line 6: ; ()  
Line 7: } ()  
Line 7: ; ()



```
0 *ast.File {
1 . Package: helloworld.go:1:1
2 . Name: *ast.Ident {
3 . . NamePos: helloworld.go:1:9
4 . . Name: "main"
5 . }
6 . Decls: []ast.Decl (len = 1) {
7 . . 0: *ast.GenDecl {
8 . . . TokPos: helloworld.go:3:1
9 . . . Tok: import
10 . . . Lparen: -
11 . . . Specs: []ast.Spec (len = 1) {
12 . . . . 0: *ast.ImportSpec {
13 . . . . . Path: *ast.BasicLit {
14 . . . . . . ValuePos: helloworld.go:3:8
15 . . . . . . Kind: STRING
16 . . . . . . Value: "\"fmt\""
17 . . . . . }
18 . . . . . EndPos: -
19 . . . . }
20 . . . }
21 . . . Rparen: -
22 . . }
23 . }
24 . Scope: *ast.Scope {
25 . . Imports: []*ast.ImportSpec (len = 1) {
26 . . . 0: *(obj @ 12)
27 . . . }
28 . . Unresolved: []*ast.Ident (len = 0) {
29 . . . }
30 . }
```

# SAMPLE PARSER FUNCTIONS

```
// ImportSpec = [ "." | PackageName ] ImportPath .
// ImportPath = string_lit .

func (p *parser) importDecl (group *Group) Decl {
    if trace {
        defer p.trace("importDecl") ()
    }

    d := new(ImportDecl)
    d.pos = p.pos()
    d.Group = group
    d.Pragma = p.takePragma ()

    switch p.tok {
    case _Name:
        d.LocalPkgName = p.name ()
    case _Dot:
        d.LocalPkgName = NewName (p.pos (), ".")
        p.next ()
    }
    d.Path = p.oliteral ()
    if d.Path == nil {
        p.syntaxError ("missing import path" )
        p.advance (_Semi, _Rparen)
        return d
    }
    if !d.Path.Bad && d.Path.Kind != StringLit {
        p.syntaxError ("import path must be a string" )
        d.Path.Bad = true
    }
    // d.Path.Bad || d.Path.Kind == StringLit

    return d
}
```

# SAMPLE PARSER FUNCTIONS

```
// ImportSpec = [ "." | PackageName ] ImportPath .
// ImportPath = string_lit .

func (p *parser) importDecl (group *Group) Decl {
    if trace {
        defer p.trace("importDecl") ()
    }

    d := new(ImportDecl)
    d.pos = p.pos()
    d.Group = group
    d.Pragma = p.takePragma ()

    switch p.tok {
    case _Name:
        d.LocalPkgName = p.name ()
    case _Dot:
        d.LocalPkgName = NewName (p.pos (), ".")
        p.next ()
    }
    d.Path = p.oliteral ()
    if d.Path == nil {
        p.syntaxError ("missing import path" )
        p.advance (_Semi, _Rparen)
        return d
    }
    if !d.Path.Bad && d.Path.Kind != StringLit {
        p.syntaxError ("import path must be a string" )
        d.Path.Bad = true
    }
    // d.Path.Bad || d.Path.Kind == StringLit

    return d
}
```

# SAMPLE PARSER FUNCTIONS

```
// ImportSpec = [ "." | PackageName ] ImportPath .
// ImportPath = string_lit .

func (p *parser) importDecl (group *Group) Decl {
    if trace {
        defer p.trace("importDecl") ()
    }

    d := new(ImportDecl)
    d.pos = p.pos()
    d.Group = group
    d.Pragma = p.takePragma()

    switch p.tok {
    case _Name:
        d.LocalPkgName = p.name()
    case _Dot:
        d.LocalPkgName = NewName(p.pos(), ".")
        p.next()
    }
    d.Path = p.oliteral()
    if d.Path == nil {
        p.syntaxError("missing import path")
        p.advance(_Semi, _Rparen)
        return d
    }
    if !d.Path.Bad && d.Path.Kind != StringLit {
        p.syntaxError("import path must be a string")
        d.Path.Bad = true
    }
    // d.Path.Bad || d.Path.Kind == StringLit

    return d
}
```



# SAMPLE PARSER FUNCTIONS

```
// ImportSpec = [ "." | PackageName ] ImportPath .
// ImportPath = string_lit .

func (p *parser) importDecl (group *Group) Decl {
    if trace {
        defer p.trace("importDecl") ()
    }

    d := new(ImportDecl)
    d.pos = p.pos()
    d.Group = group
    d.Pragma = p.takePragma()

    switch p.tok {
    case _Name:
        d.LocalPkgName = p.name()
    case _Dot:
        d.LocalPkgName = NewName(p.pos(), ".")
        p.next()
    }

    d.Path = p.oliteral()
    if d.Path == nil {
        p.syntaxError("missing import path")
        p.advance(_Semi, _Rparen)
        return d
    }

    if !d.Path.Bad && d.Path.Kind != StringLit {
        p.syntaxError("import path must be a string")
        d.Path.Bad = true
    }

    // d.Path.Bad || d.Path.Kind == StringLit

    return d
}
```

# SAMPLE PARSER FUNCTIONS

```
// ImportSpec = [ "." | PackageName ] ImportPath .
// ImportPath = string_lit .

func (p *parser) importDecl (group *Group) Decl {
    if trace {
        defer p.trace("importDecl") ()
    }

    d := new(ImportDecl)
    d.pos = p.pos()
    d.Group = group
    d.Pragma = p.takePragma ()

    switch p.tok {
    case _Name:
        d.LocalPkgName = p.name ()
    case _Dot:
        d.LocalPkgName = NewName (p.pos (), ".")
        p.next ()
    }
    d.Path = p.oliteral ()
    if d.Path == nil {
        p.syntaxError ("missing import path" )
        p.advance ( _Semi, _Rparen )
        return d
    }
    if !d.Path.Bad && d.Path.Kind != StringLit {
        p.syntaxError ("import path must be a string" )
        d.Path.Bad = true
    }
    // d.Path.Bad || d.Path.Kind == StringLit

    return d
}
```

# SAMPLE PARSER FUNCTIONS

```
// ImportSpec = [ "." | PackageName ] ImportPath .
// ImportPath = string_lit .

func (p *parser) importDecl (group *Group) Decl {
    if trace {
        defer p.trace("importDecl") ()
    }

    d := new(ImportDecl)
    d.pos = p.pos()
    d.Group = group
    d.Pragma = p.takePragma ()

    switch p.tok {
    case _Name:
        d.LocalPkgName = p.name ()
    case _Dot:
        d.LocalPkgName = NewName (p.pos (), ".")
        p.next ()
    }
    d.Path = p.oliteral ()
    if d.Path == nil {
        p.syntaxError ("missing import path" )
        p.advance (_Semi, _Rparen)
        return d
    }
    if !d.Path.Bad && d.Path.Kind != StringLit {
        p.syntaxError ("import path must be a string" )
        d.Path.Bad = true
    }
    // d.Path.Bad || d.Path.Kind == StringLit

    return d
}
```

# SAMPLE PARSER FUNCTIONS

```
// FunctionDecl = "func" FunctionName [ TypeParams ] ( Function | Signature ) .
// FunctionName = identifier .
// Function      = Signature FunctionBody .
// MethodDecl    = "func" Receiver MethodName ( Function | Signature ) .
// Receiver      = Parameters .
func (p *parser) funcDeclOrNil () *FuncDecl {
    if trace {
        defer p.trace("funcDecl") ()
    }

    f := new(FuncDecl)
    f.pos = p.pos()
    f.Pragma = p.takePragma ()

    if p.got(_Lparen) {
        rcvr := p.paramList(nil, nil, _Rparen, false)
        switch len(rcvr) {
        case 0:
            p.error("method has no receiver")
        default:
            p.error("method has multiple receivers")
            fallthrough
        case 1:
            f.Recv = rcvr[0]
        }
    }

    if p.tok != _Name {
        p.syntaxError("expecting name or (")
        p.advance(_Lbrace, _Semi)
        return nil
    }

    f.Name = p.name()

    context := ""
    if f.Recv != nil {
        context = "method" // don't permit (method) type parameters in funcType
    }
    f.TParamList, f.Type = p.funcType(context)

    if p.tok == _Lbrace {
        f.Body = p.funcBody()
    }

    return f
}
```

# SAMPLE PARSER FUNCTIONS

```
// FunctionDecl = "func" FunctionName [ TypeParams ] ( Function | Signature ) .
// FunctionName = identifier .
// Function      = Signature FunctionBody .
// MethodDecl    = "func" Receiver MethodName ( Function | Signature ) .
// Receiver      = Parameters .

func (p *parser) funcDeclOrNil () *FuncDecl {
    if trace {
        defer p.trace("funcDecl") ()
    }

    f := new(FuncDecl)
    f.pos = p.pos()
    f.Pragma = p.takePragma ()

    if p.got(_Lparen) {
        rcvr := p.paramList(nil, nil, _Rparen, false)
        switch len(rcvr) {
        case 0:
            p.error("method has no receiver")
        default:
            p.error("method has multiple receivers")
            fallthrough
        case 1:
            f.Recv = rcvr[0]
        }
    }

    if p.tok != _Name {
        p.syntaxError("expecting name or (")
        p.advance(_Lbrace, _Semi)
        return nil
    }

    f.Name = p.name()

    context := ""
    if f.Recv != nil {
        context = "method" // don't permit (method) type parameters in funcType
    }
    f.TParamList, f.Type = p.funcType(context)

    if p.tok == _Lbrace {
        f.Body = p.funcBody()
    }

    return f
}
```

# SAMPLE PARSER FUNCTIONS

```
// FunctionDecl = "func" FunctionName [ TypeParams ] ( Function | Signature ) .
// FunctionName = identifier .
// Function      = Signature FunctionBody .
// MethodDecl    = "func" Receiver MethodName ( Function | Signature ) .
// Receiver      = Parameters .
func (p *parser) funcDeclOrNil () *FuncDecl {
    if trace {
        defer p.trace("funcDecl") ()
    }

    f := new(FuncDecl)
    f.pos = p.pos()
    f.Pragma = p.takePragma ()

    if p.got(_Lparen) {
        rcvr := p.paramList(nil, nil, _Rparen, false)
        switch len(rcvr) {
        case 0:
            p.error("method has no receiver")
        default:
            p.error("method has multiple receivers")
            fallthrough
        case 1:
            f.Recv = rcvr[0]
        }
    }

    if p.tok != _Name {
        p.syntaxError("expecting name or (")
        p.advance(_Lbrace, _Semi)
        return nil
    }

    f.Name = p.name()

    context := ""
    if f.Recv != nil {
        context = "method" // don't permit (method) type parameters in funcType
    }
    f.TParamList, f.Type = p.funcType(context)

    if p.tok == _Lbrace {
        f.Body = p.funcBody()
    }

    return f
}
```

# SAMPLE PARSER FUNCTIONS

```
// FunctionDecl = "func" FunctionName [ TypeParams ] ( Function | Signature ) .
// FunctionName = identifier .
// Function      = Signature FunctionBody .
// MethodDecl    = "func" Receiver MethodName ( Function | Signature ) .
// Receiver      = Parameters .
func (p *parser) funcDeclOrNil () *FuncDecl {
    if trace {
        defer p.trace("funcDecl") ()
    }

    f := new(FuncDecl)
    f.pos = p.pos()
    f.Pragma = p.takePragma ()

    if p.got( Lparen ) {
        rcvr := p.paramList (nil, nil, _Rparen, false)
        switch len(rcvr) {
        case 0:
            p.error("method has no receiver" )
        default:
            p.error("method has multiple receivers" )
            fallthrough
        case 1:
            f.Recv = rcvr[0]
        }
    }

    if p.tok != _Name {
        p.syntaxError("expecting name or (")
        p.advance(_Lbrace, _Semi)
        return nil
    }

    f.Name = p.name()

    context := ""
    if f.Recv != nil {
        context = "method" // don't permit (method) type parameters in funcType
    }
    f.TParamList, f.Type = p.funcType(context)

    if p.tok == _Lbrace {
        f.Body = p.funcBody()
    }

    return f
}
```



# SAMPLE PARSER FUNCTIONS

```
// FunctionDecl = "func" FunctionName [ TypeParams ] ( Function | Signature ) .
// FunctionName = identifier .
// Function      = Signature FunctionBody .
// MethodDecl    = "func" Receiver MethodName ( Function | Signature ) .
// Receiver      = Parameters .
func (p *parser) funcDeclOrNil () *FuncDecl {
    if trace {
        defer p.trace("funcDecl") ()
    }

    f := new(FuncDecl)
    f.pos = p.pos()
    f.Pragma = p.takePragma ()

    if p.got(_Lparen) {
        rcvr := p.paramList(nil, nil, _Rparen, false)
        switch len(rcvr) {
        case 0:
            p.error("method has no receiver")
        default:
            p.error("method has multiple receivers")
            fallthrough
        case 1:
            f.Recv = rcvr[0]
        }
    }

    if p.tok != Name {
        p.syntaxError("expecting name or (")
        p.advance(_Lbrace, _Semi)
        return nil
    }

    f.Name = p.name()

    context := ""
    if f.Recv != nil {
        context = "method" // don't permit (method) type parameters in funcType
    }
    f.TParamList, f.Type = p.funcType(context)

    if p.tok == _Lbrace {
        f.Body = p.funcBody()
    }

    return f
}
```

# SAMPLE PARSER FUNCTIONS

```
// FunctionDecl = "func" FunctionName [ TypeParams ] ( Function | Signature ) .
// FunctionName = identifier .
// Function      = Signature FunctionBody .
// MethodDecl    = "func" Receiver MethodName ( Function | Signature ) .
// Receiver      = Parameters .
func (p *parser) funcDeclOrNil () *FuncDecl {
    if trace {
        defer p.trace("funcDecl") ()
    }

    f := new(FuncDecl)
    f.pos = p.pos()
    f.Pragma = p.takePragma ()

    if p.got(_Lparen) {
        rcvr := p.paramList(nil, nil, _Rparen, false)
        switch len(rcvr) {
        case 0:
            p.error("method has no receiver")
        default:
            p.error("method has multiple receivers")
            fallthrough
        case 1:
            f.Recv = rcvr[0]
        }
    }

    if p.tok != _Name {
        p.syntaxError("expecting name or (")
        p.advance(_Lbrace, _Semi)
        return nil
    }

    f.Name = p.name()

    context := ""
    if f.Recv != nil {
        context = "method" // don't permit (method) type parameters in funcType
    }
    f.TParamList, f.Type = p.funcType(context)

    if p.tok == _Lbrace {
        f.Body = p.funcBody()
    }

    return f
}
```

# SAMPLE PARSER FUNCTIONS

```
// FunctionDecl = "func" FunctionName [ TypeParams ] ( Function | Signature ) .
// FunctionName = identifier .
// Function      = Signature FunctionBody .
// MethodDecl    = "func" Receiver MethodName ( Function | Signature ) .
// Receiver      = Parameters .
func (p *parser) funcDeclOrNil () *FuncDecl {
    if trace {
        defer p.trace("funcDecl") ()
    }

    f := new(FuncDecl)
    f.pos = p.pos()
    f.Pragma = p.takePragma ()

    if p.got(_Lparen) {
        rcvr := p.paramList(nil, nil, _Rparen, false)
        switch len(rcvr) {
        case 0:
            p.error("method has no receiver")
        default:
            p.error("method has multiple receivers")
            fallthrough
        case 1:
            f.Recv = rcvr[0]
        }
    }

    if p.tok != _Name {
        p.syntaxError("expecting name or (")
        p.advance(_Lbrace, _Semi)
        return nil
    }

    f.Name = p.name()

    context := ""
    if f.Recv != nil {
        context = "method" // don't permit (method) type parameters in funcType
    }
    f.TParamList, f.Type = p.funcType(context)

    if p.tok == Lbrace {
        f.Body = p.funcBody()
    }

    return f
}
```

# SAMPLE PARSER FUNCTIONS

```
// FunctionDecl = "func" FunctionName [ TypeParams ] ( Function | Signature ) .
// FunctionName = identifier .
// Function      = Signature FunctionBody .
// MethodDecl    = "func" Receiver MethodName ( Function | Signature ) .
// Receiver      = Parameters .
func (p *parser) funcDeclOrNil () *FuncDecl {
    if trace {
        defer p.trace("funcDecl") ()
    }

    f := new(FuncDecl)
    f.pos = p.pos()
    f.Pragma = p.takePragma ()

    if p.got(_Lparen) {
        rcvr := p.paramList(nil, nil, _Rparen, false)
        switch len(rcvr) {
        case 0:
            p.error("method has no receiver")
        default:
            p.error("method has multiple receivers")
            fallthrough
        case 1:
            f.Recv = rcvr[0]
        }
    }

    if p.tok != _Name {
        p.syntaxError("expecting name or (")
        p.advance(_Lbrace, _Semi)
        return nil
    }

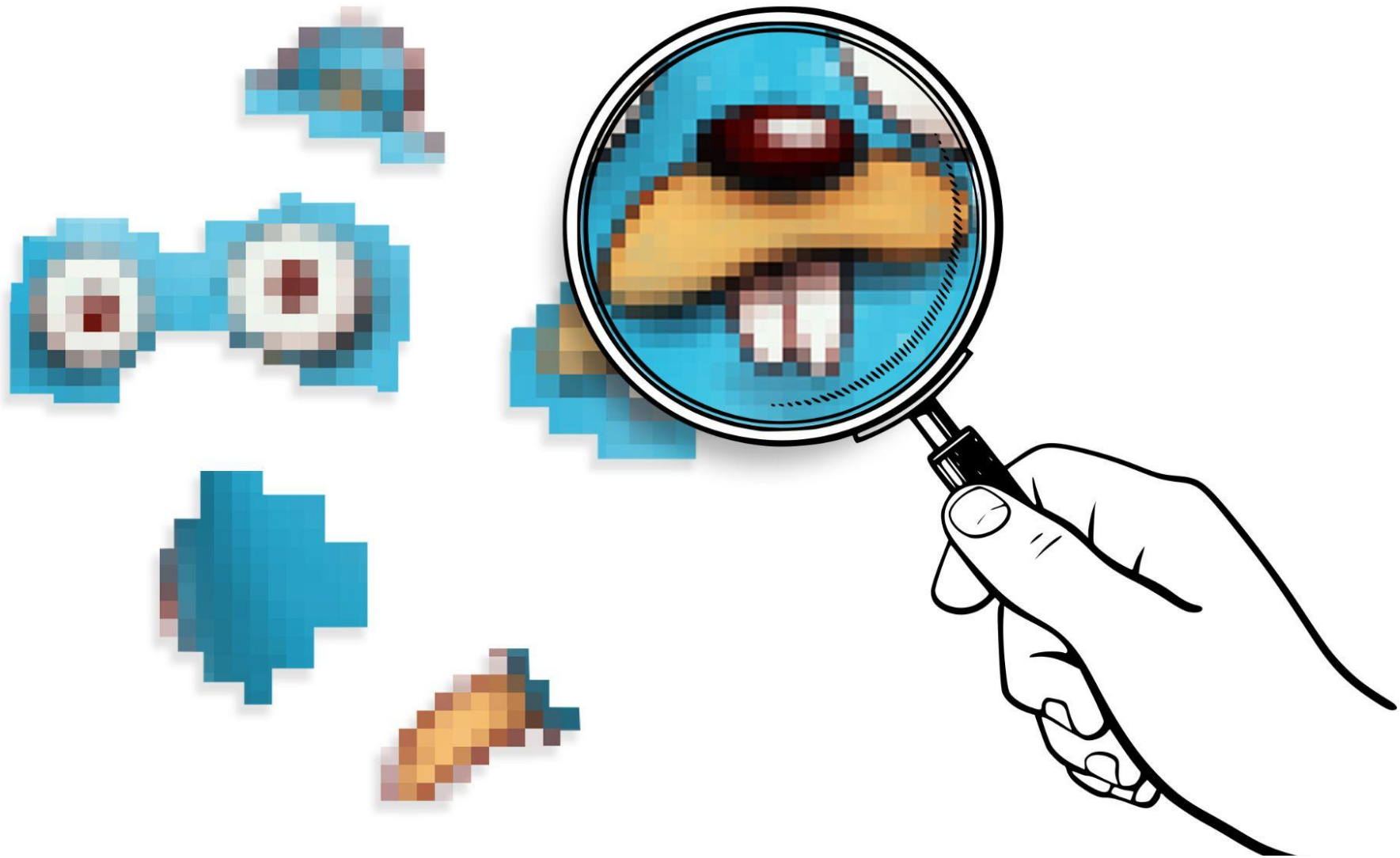
    f.Name = p.name()

    context := ""
    if f.Recv != nil {
        context = "method" // don't permit (method) type parameters in funcType
    }
    f.TParamList, f.Type = p.funcType(context)

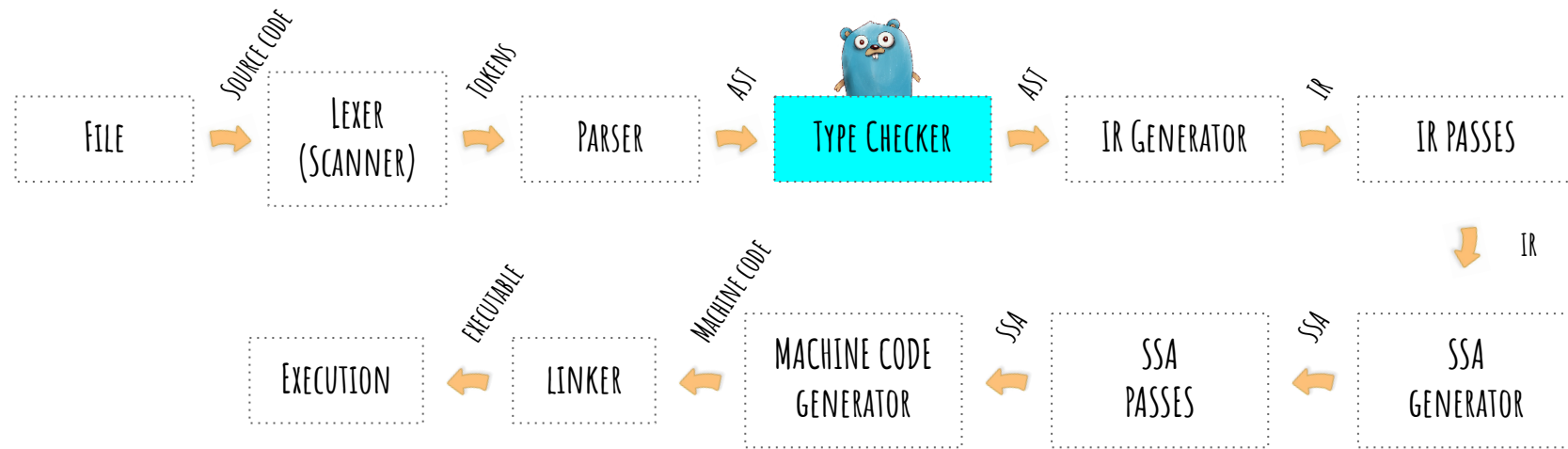
    if p.tok == _Lbrace {
        f.Body = p.funcBody()
    }

    return f
}
```

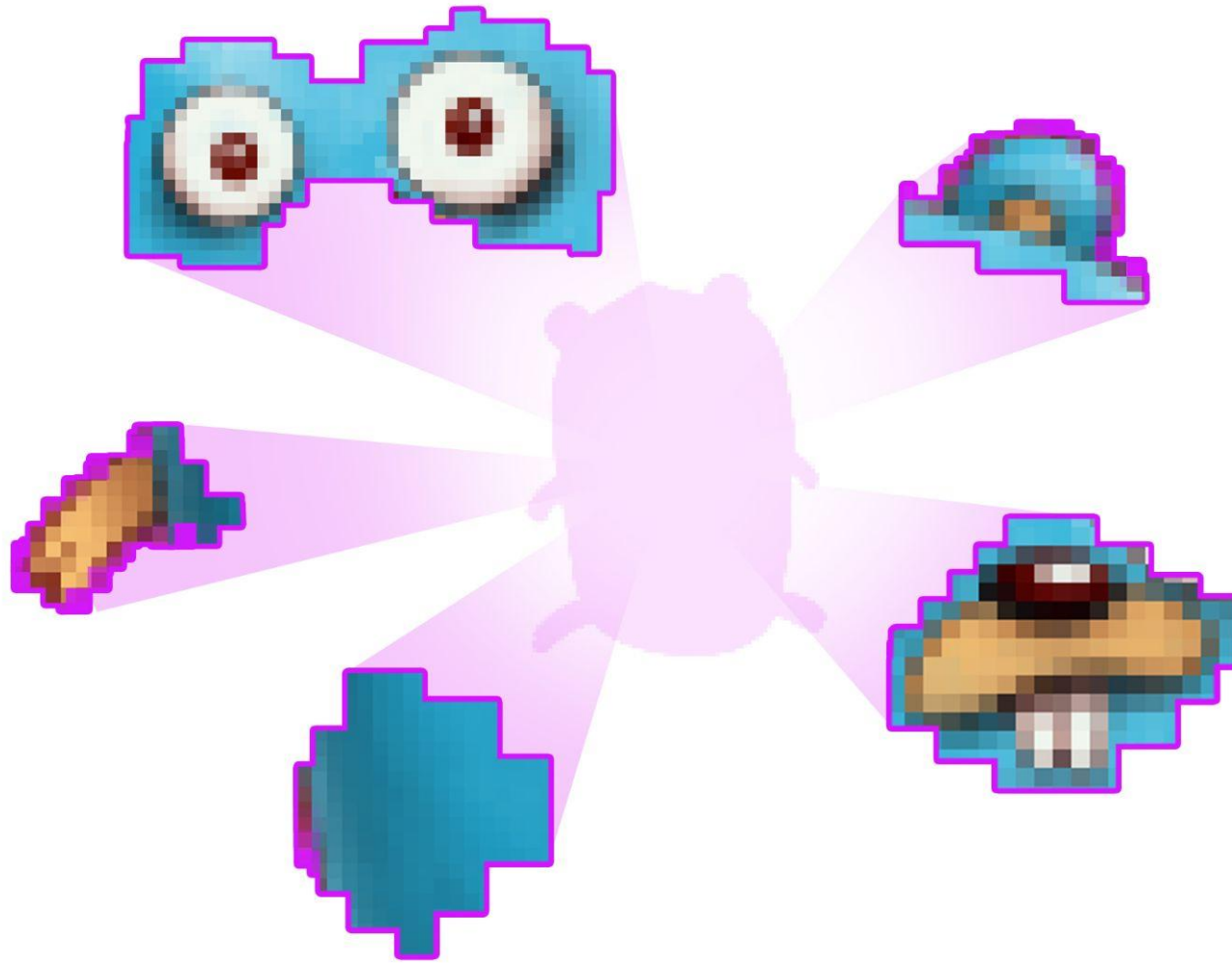
# TYPE CHECKER



# TYPE CHECKER

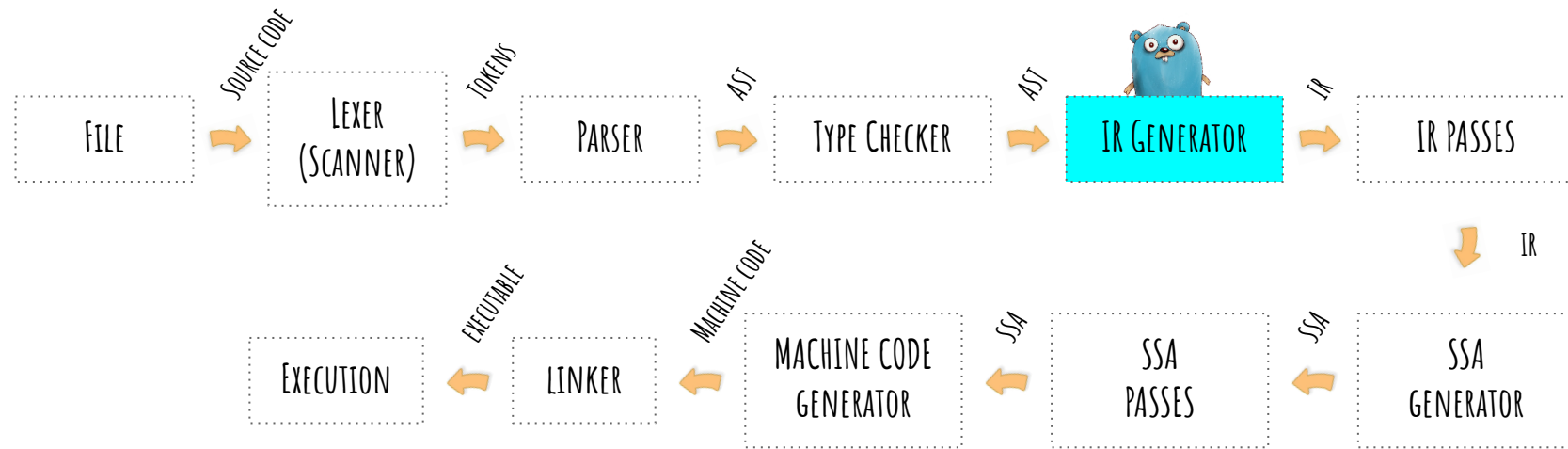


# THE IR





# THE IR

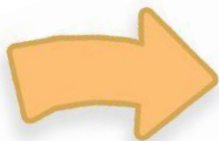


# THE IR

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



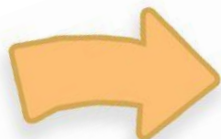
```
1  *ir.Package {  
2  .   Imports: []*types.Pkg (1 entries) {  
3  .   .   0: *types.Pkg {  
4  .   .   .   Path: "fmt"  
5  .   .   .   Name: "fmt"  
6  .   .   .   Prefix: "fmt"  
7  .   .   .   Syms: map[.inittask:fmt..inittask Append:fmt.Append ...]  
8  .   .   .   Height: 10  
9  .   .   .   Direct: true  
10 .   .   }  
11 .   }  
12 . }  
13 . Decls: []ir.Node (1 entries) {  
14 . .   0: *ir.Func {  
15 . . .   Body: ir.Nodes (1 entries) {  
16 . . . .   0: *ir.CallExpr {  
17 . . . . .   X: *ir.Name {  
18 . . . . . .   Class: PFUNC  
19 . . . . . .   Func: *ir.Func {  
20 . . . . . . .   Nname: *(@17)  
21 . . . . . . .   FieldTrack: map[]  
22 . . . . . . .   Inl: *ir.Inline {  
23 . . . . . . . .   Cost: 72  
24 . . . . . . . }  
25 . . . . . . }  
26 . . . . . Endlineno: $GOROOT/src/fmt/print.go:295:1  
27 . . . . . WBPpos: <unknown line number >  
28 . . . . . ABI: ABIInternal  
29 . . . . }  
30 . . . }  
31 . . }  
32 . . }  
33 . . .   Args: ir.Nodes (1 entries) {  
34 . . . .   0: *ir.ConvExpr {  
35 . . . . .   X: *ir.ConstExpr {}  
36 . . . . }  
37 . . . }  
38 . . }  
39 . . }  
40 . . }  
41 . . .   Nname: *ir.Name {  
42 . . . .   Class: PFUNC  
43 . . . . .   Func: *(@14)  
44 . . . . .   Defn: *(@14)  
45 . . . . }  
46 . . . }  
47 . . .   Parents: []ir.ScopeID (0 entries) {}  
48 . . .   Marks: []ir.Mark (0 entries) {}  
49 . . .   FieldTrack: map[]  
50 . . .   Endlineno: ../hello.go:7:1  
51 . . .   WBPpos: <unknown line number >  
52 . . .   ABI: ABIInternal  
53 . . }  
54 . }  
55 . }  
56 . }  
57 }
```

# THE IR

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



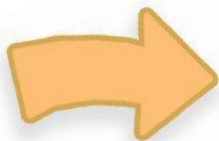
```
1  *ir.Package {  
2  . Imports: []*types.Pkg (1 entries) {  
3  . . 0: *types.Pkg {  
4  . . . Path: "fmt"  
5  . . . Name: "fmt"  
6  . . . Prefix: "fmt"  
7  . . . Syms: map[.inittask:fmt..inittask Append:fmt.Append ...]  
8  . . . Height: 10  
9  . . . Direct: true  
10 . . . ...  
11 . . }  
12 . }  
13 . Decls: []ir.Node (1 entries) {  
14 . . 0: *ir.Func {  
15 . . . Body: ir.Nodes (1 entries) {  
16 . . . . 0: *ir.CallExpr {  
17 . . . . . X: *ir.Name {  
18 . . . . . . Class: PFUNC  
19 . . . . . . Func: *ir.Func {  
20 . . . . . . . Nname: *(@17)  
21 . . . . . . . FieldTrack: map[]  
22 . . . . . . . Inl: *ir.Inline {  
23 . . . . . . . . Cost: 72  
24 . . . . . . . ...  
25 . . . . . . }  
26 . . . . . . Endlineno: $GOROOT/src/fmt/print.go:295:1  
27 . . . . . . WBPos: <unknown line number >  
28 . . . . . . ABI: ABIInternal  
29 . . . . . . ...  
30 . . . . . }  
31 . . . . . ...  
32 . . . . . }  
33 . . . . . Args: ir.Nodes (1 entries) {  
34 . . . . . . 0: *ir.ConvExpr {  
35 . . . . . . . X: *ir.ConstExpr {}  
36 . . . . . . }  
37 . . . . . }  
38 . . . . . ...  
39 . . . . . }  
40 . . . . . }  
41 . . . . . Nname: *ir.Name {  
42 . . . . . . Class: PFUNC  
43 . . . . . . Func: *(@14)  
44 . . . . . . Defn: *(@14)  
45 . . . . . . ...  
46 . . . . . }  
47 . . . . . Parents: []ir.ScopeID (0 entries) {}  
48 . . . . . Marks: []ir.Mark (0 entries) {}  
49 . . . . . FieldTrack: map[]  
50 . . . . . Endlineno: ../hello.go:7:1  
51 . . . . . WBPos: <unknown line number >  
52 . . . . . ABI: ABIInternal  
53 . . . . . ...  
54 . . . . . }  
55 . . }  
56 . ...  
57 }
```

# THE IR

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



```
1  *ir.Package {  
2  .   Imports: []*types.Pkg (1 entries) {  
3  .   .   0: *types.Pkg {  
4  .   .   .   Path: "fmt"  
5  .   .   .   Name: "fmt"  
6  .   .   .   Prefix: "fmt"  
7  .   .   .   Syms: map[.inittask:fmt..inittask Append:fmt.Append ...]  
8  .   .   .   Height: 10  
9  .   .   .   Direct: true  
10 .   .   ...  
11 .   .   }  
12 .   }  
13 .   Decls: []ir.Node (1 entries) {  
14 .   .   0: *ir.Func {  
15 .   .   .   Body: ir.Nodes (1 entries) {  
16 .   .   .   .   0: *ir.CallExpr {  
17 .   .   .   .   .   X: *ir.Name {  
18 .   .   .   .   .   .   Class: PFUNC  
19 .   .   .   .   .   .   Func: *ir.Func {  
20 .   .   .   .   .   .   .   Nname: *(@17)  
21 .   .   .   .   .   .   .   FieldTrack: map[]  
22 .   .   .   .   .   .   .   Inl: *ir.Inline {  
23 .   .   .   .   .   .   .   .   Cost: 72  
24 .   .   .   .   .   .   .   ...  
25 .   .   .   .   .   .   }  
26 .   .   .   .   .   .   Endlineno: $GOROOT/src/fmt/print.go:295:1  
27 .   .   .   .   .   .   WbPos: <unknown line number >  
28 .   .   .   .   .   .   ABI: ABIInternal  
29 .   .   .   .   .   .   ...  
30 .   .   .   .   .   }  
31 .   .   .   .   .   ...  
32 .   .   .   .   }  
33 .   .   .   .   Args: ir.Nodes (1 entries) {  
34 .   .   .   .   .   0: *ir.ConvExpr {  
35 .   .   .   .   .   .   X: *ir.ConstExpr {}  
36 .   .   .   .   .   }  
37 .   .   .   .   }  
38 .   .   .   .   ...  
39 .   .   .   .   }  
40 .   .   .   }  
41 .   .   .   Nname: *ir.Name {  
42 .   .   .   .   Class: PFUNC  
43 .   .   .   .   Func: *(@14)  
44 .   .   .   .   Defn: *(@14)  
45 .   .   .   .   ...  
46 .   .   .   }  
47 .   .   .   Parents: []ir.ScopeID (0 entries) {}  
48 .   .   .   Marks: []ir.Mark (0 entries) {}  
49 .   .   .   FieldTrack: map[]  
50 .   .   .   Endlineno: ../hello.go:7:1  
51 .   .   .   WbPos: <unknown line number >  
52 .   .   .   ABI: ABIInternal  
53 .   .   .   ...  
54 .   .   }  
55 .   }  
56 .   ...  
57 }
```

# THE IR

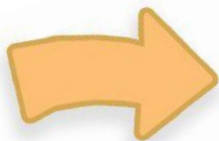
```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    fmt.Println("hello world!")
```

```
}
```



```
1  *ir.Package {
2  .   Imports: []*types.Pkg (1 entries) {
3  .   .   0: *types.Pkg {
4  .   .   .   Path: "fmt"
5  .   .   .   Name: "fmt"
6  .   .   .   Prefix: "fmt"
7  .   .   .   Syms: map[.inittask:fmt..inittask Append:fmt.Append ...]
8  .   .   .   Height: 10
9  .   .   .   Direct: true
10 .   .   ...
11 .   .   }
12 .   }
13 .   Decls: []ir.Node (1 entries) {
14 .   .   0: *ir.Func {
15 .   .   .   Body: ir.Nodes (1 entries) {
16 .   .   .   .   0: *ir.CallExpr {
17 .   .   .   .   .   X: *ir.Name {
18 .   .   .   .   .   .   Class: PFUNC
19 .   .   .   .   .   .   Func: *ir.Func {
20 .   .   .   .   .   .   .   Nname: *(@17)
21 .   .   .   .   .   .   .   FieldTrack: map[]
22 .   .   .   .   .   .   .   Inl: *ir.Inline {
23 .   .   .   .   .   .   .   .   Cost: 72
24 .   .   .   .   .   .   .   ...
25 .   .   .   .   .   .   }
26 .   .   .   .   .   .   Endlineno: $GOROOT/src/fmt/print.go:295:1
27 .   .   .   .   .   .   WBPos: <unknown line number >
28 .   .   .   .   .   .   ABI: ABIInternal
29 .   .   .   .   .   .   ...
30 .   .   .   .   .   }
31 .   .   .   .   .   ...
32 .   .   .   .   }
33 .   .   .   .   Args: ir.Nodes (1 entries) {
34 .   .   .   .   .   0: *ir.ConvExpr {
35 .   .   .   .   .   .   X: *ir.ConstExpr {}
36 .   .   .   .   .   }
37 .   .   .   .   }
38 .   .   .   .   ...
39 .   .   .   }
40 .   .   }
41 .   .   .   Nname: *ir.Name {
42 .   .   .   .   Class: PFUNC
43 .   .   .   .   Func: *(@14)
44 .   .   .   .   Defn: *(@14)
45 .   .   .   .   ...
46 .   .   .   }
47 .   .   .   Parents: []ir.ScopeID (0 entries) {}
48 .   .   .   Marks: []ir.Mark (0 entries) {}
49 .   .   .   FieldTrack: map[]
50 .   .   .   Endlineno: ../hello.go:7:1
51 .   .   .   WBPos: <unknown line number >
52 .   .   .   ABI: ABIInternal
53 .   .   .   ...
54 .   .   }
55 .   }
56 .   ...
57 }
```

# THE IR GENERATION FUNCTIONS

```
func (g *irgen) constDecl(out *ir.Nodes, decl *syntax.ConstDecl) {
    g.pragmaFlags(decl.Pragma, 0)

    for _, name := range decl.NameList {
        name, obj := g.def(name)

        // For untyped numeric constants, make sure the value
        // representation matches what the rest of the
        // compiler (really just iexport) expects.
        // TODO(mdempsky): Revisit after #43891 is resolved.
        val := obj.(*types2.Const).Val()
        switch name.Type() {
        case types.UntypedInt, types.UntypedRune:
            val = constant.ToInt(val)
        case types.UntypedFloat:
            val = constant.ToFloat(val)
        case types.UntypedComplex:
            val = constant.ToComplex(val)
        }
        name.SetVal(val)

        out.Append(ir.NewDecl(g.pos(decl), ir.ODCLCONST, name))
    }
}
```

```
func (g *irgen) funcDecl(out *ir.Nodes, decl *syntax.FuncDecl) {
    ... // Omitted code

    fn := ir.NewFunc(g.pos(decl))
    fn.Nname, _ = g.def(decl.Name)
    fn.Nname.Func = fn
    fn.Nname.Defn = fn

    ... // Omitted code

    if decl.Name.Value == "init" && decl.Recv == nil {
        g.target.Inits = append(g.target.Inits, fn)
    }

    saveHaveEmbed := g.haveEmbed
    saveCurDecl := g.curDecl
    g.curDecl = ""
    g.later(func() {
        defer func(b bool, s string) {
            g.haveEmbed = b
            g.curDecl = s
        }(g.haveEmbed, g.curDecl)

        g.haveEmbed = saveHaveEmbed
        g.curDecl = saveCurDecl
        if fn.Type().HasTParam() {
            g.topFuncIsGeneric = true
        }
        g.funcBody(fn, decl.Recv, decl.Type, decl.Body)
        g.topFuncIsGeneric = false
        if fn.Type().HasTParam() && fn.Body != nil {
            fn.Inl = &ir.Inline{
                Cost: 1,
                Dcl:  fn.Dcl,
                Body: fn.Body,
            }
        }
    })
    out.Append(fn)
}
```

# THE IR GENERATION FUNCTIONS

```
func (g *irgen) constDecl(out *ir.Nodes, decl *syntax.ConstDecl) {
    g.pragmaFlags(decl.Pragma, 0)

    for _, name := range decl.NameList {
        name, obj := g.def(name)

        // For untyped numeric constants, make sure the value
        // representation matches what the rest of the
        // compiler (really just iexport) expects.
        // TODO(mdempsky): Revisit after #43891 is resolved.
        val := obj.(*types2.Const).Val()
        switch name.Type() {
        case types.UntypedInt, types.UntypedRune:
            val = constant.ToInt(val)
        case types.UntypedFloat:
            val = constant.ToFloat(val)
        case types.UntypedComplex:
            val = constant.ToComplex(val)
        }
        name.SetVal(val)

        out.Append(ir.NewDecl(g.pos(decl), ir.ODCLCONST, name))
    }
}
```

```
func (g *irgen) funcDecl(out *ir.Nodes, decl *syntax.FuncDecl) {
    ... // Omitted code

    fn := ir.NewFunc(g.pos(decl))
    fn.Nname, _ = g.def(decl.Name)
    fn.Nname.Func = fn
    fn.Nname.Defn = fn

    ... // Omitted code

    if decl.Name.Value == "init" && decl.Recv == nil {
        g.target.Inits = append(g.target.Inits, fn)
    }

    saveHaveEmbed := g.haveEmbed
    saveCurDecl := g.curDecl
    g.curDecl = ""
    g.later(func() {
        defer func(b bool, s string) {
            g.haveEmbed = b
            g.curDecl = s
        }(g.haveEmbed, g.curDecl)

        g.haveEmbed = saveHaveEmbed
        g.curDecl = saveCurDecl
        if fn.Type().HasTParam() {
            g.topFuncIsGeneric = true
        }
        g.funcBody(fn, decl.Recv, decl.Type, decl.Body)
        g.topFuncIsGeneric = false
        if fn.Type().HasTParam() && fn.Body != nil {
            fn.Inl = &ir.Inline{
                Cost: 1,
                Dcl:  fn.Dcl,
                Body: fn.Body,
            }
        }
    })
    out.Append(fn)
}
```

# THE IR GENERATION FUNCTIONS

```
func (g *irgen) constDecl(out *ir.Nodes, decl *syntax.ConstDecl) {
    g.pragmaFlags(decl.Pragma, 0)

    for _, name := range decl.NameList {
        name, obj := g.def(name)

        // For untyped numeric constants, make sure the value
        // representation matches what the rest of the
        // compiler (really just iexport) expects.
        // TODO(mdempsky): Revisit after #43891 is resolved.
        val := obj.(*types2.Const).Val()
        switch name.Type() {
        case types.UntypedInt, types.UntypedRune:
            val = constant.ToInt(val)
        case types.UntypedFloat:
            val = constant.ToFloat(val)
        case types.UntypedComplex:
            val = constant.ToComplex(val)
        }
        name.SetVal(val)

        out.Append(ir.NewDecl(g.pos(decl), ir.ODCLCONST, name))
    }
}
```

```
func (g *irgen) funcDecl(out *ir.Nodes, decl *syntax.FuncDecl) {
    ... // Omitted code

    fn := ir.NewFunc(g.pos(decl))
    fn.Nname, _ = g.def(decl.Name)
    fn.Nname.Func = fn
    fn.Nname.Defn = fn

    ... // Omitted code

    if decl.Name.Value == "init" && decl.Recv == nil {
        g.target.Inits = append(g.target.Inits, fn)
    }

    saveHaveEmbed := g.haveEmbed
    saveCurDecl := g.curDecl
    g.curDecl = ""
    g.later(func() {
        defer func(b bool, s string) {
            g.haveEmbed = b
            g.curDecl = s
        }(g.haveEmbed, g.curDecl)

        g.haveEmbed = saveHaveEmbed
        g.curDecl = saveCurDecl
        if fn.Type().HasTParam() {
            g.topFuncIsGeneric = true
        }
        g.funcBody(fn, decl.Recv, decl.Type, decl.Body)
        g.topFuncIsGeneric = false
        if fn.Type().HasTParam() && fn.Body != nil {
            fn.Inl = &ir.Inline{
                Cost: 1,
                Dcl:  fn.Dcl,
                Body: fn.Body,
            }
        }
        out.Append(fn)
    })
}
```



# THE IR GENERATION FUNCTIONS

```
func (g *irgen) constDecl(out *ir.Nodes, decl *syntax.ConstDecl) {
    g.pragmaFlags(decl.Pragma, 0)

    for _, name := range decl.NameList {
        name, obj := g.def(name)

        // For untyped numeric constants, make sure the value
        // representation matches what the rest of the
        // compiler (really just iexport) expects.
        // TODO(mdempsky): Revisit after #43891 is resolved.
        val := obj.(*types2.Const).Val()
        switch name.Type() {
        case types.UntypedInt, types.UntypedRune:
            val = constant.ToInt(val)
        case types.UntypedFloat:
            val = constant.ToFloat(val)
        case types.UntypedComplex:
            val = constant.ToComplex(val)
        }
        name.SetVal(val)

        out.Append(ir.NewDecl(g.pos(decl), ir.ODCLCONST, name))
    }
}
```

```
func (g *irgen) funcDecl(out *ir.Nodes, decl *syntax.FuncDecl) {
    ... // Omitted code

    fn := ir.NewFunc(g.pos(decl))
    fn.Nname, _ = g.def(decl.Name)
    fn.Nname.Func = fn
    fn.Nname.Defn = fn

    ... // Omitted code

    if decl.Name.Value == "init" && decl.Recv == nil {
        g.target.Inits = append(g.target.Inits, fn)
    }

    saveHaveEmbed := g.haveEmbed
    saveCurDecl := g.curDecl
    g.curDecl = ""
    g.later(func() {
        defer func(b bool, s string) {
            g.haveEmbed = b
            g.curDecl = s
        }(g.haveEmbed, g.curDecl)

        g.haveEmbed = saveHaveEmbed
        g.curDecl = saveCurDecl
        if fn.Type().HasTParam() {
            g.topFuncIsGeneric = true
        }
        g.funcBody(fn, decl.Recv, decl.Type, decl.Body)
        g.topFuncIsGeneric = false
        if fn.Type().HasTParam() && fn.Body != nil {
            fn.Inl = &ir.Inline{
                Cost: 1,
                Dcl:  fn.Dcl,
                Body: fn.Body,
            }
        }
    })
    out.Append(fn)
}
```

# THE IR GENERATION FUNCTIONS

```
func (g *irgen) constDecl(out *ir.Nodes, decl *syntax.ConstDecl) {
    g.pragmaFlags(decl.Pragma, 0)

    for _, name := range decl.NameList {
        name, obj := g.def(name)

        // For untyped numeric constants, make sure the value
        // representation matches what the rest of the
        // compiler (really just iexport) expects.
        // TODO(mdempsky): Revisit after #43891 is resolved.
        val := obj.(*types2.Const).Val()
        switch name.Type() {
        case types.UntypedInt, types.UntypedRune:
            val = constant.ToInt(val)
        case types.UntypedFloat:
            val = constant.ToFloat(val)
        case types.UntypedComplex:
            val = constant.ToComplex(val)
        }
        name.SetVal(val)

        out.Append(ir.NewDecl(g.pos(decl), ir.ODCLCONST, name))
    }
}
```

```
func (g *irgen) funcDecl(out *ir.Nodes, decl *syntax.FuncDecl) {
    ... // Omitted code

    fn := ir.NewFunc(g.pos(decl))
    fn.Nname, _ = g.def(decl.Name)
    fn.Nname.Func = fn
    fn.Nname.Defn = fn

    ... // Omitted code

    if decl.Name.Value == "init" && decl.Recv == nil {
        g.target.Inits = append(g.target.Inits, fn)
    }

    saveHaveEmbed := g.haveEmbed
    saveCurDecl := g.curDecl
    g.curDecl = ""
    g.later(func() {
        defer func(b bool, s string) {
            g.haveEmbed = b
            g.curDecl = s
        }(g.haveEmbed, g.curDecl)

        g.haveEmbed = saveHaveEmbed
        g.curDecl = saveCurDecl
        if fn.Type().HasTParam() {
            g.topFuncIsGeneric = true
        }
        g.funcBody(fn, decl.Recv, decl.Type, decl.Body)
        g.topFuncIsGeneric = false
        if fn.Type().HasTParam() && fn.Body != nil {
            fn.Inl = &ir.Inline{
                Cost: 1,
                Dcl:  fn.Dcl,
                Body: fn.Body,
            }
        }
    })
    out.Append(fn)
}
```

# THE IR GENERATION FUNCTIONS

```
func (g *irgen) constDecl(out *ir.Nodes, decl *syntax.ConstDecl) {
    g.pragmaFlags(decl.Pragma, 0)

    for _, name := range decl.NameList {
        name, obj := g.def(name)

        // For untyped numeric constants, make sure the value
        // representation matches what the rest of the
        // compiler (really just iexport) expects.
        // TODO(mdempsky): Revisit after #43891 is resolved.
        val := obj.(*types2.Const).Val()
        switch name.Type() {
        case types.UntypedInt, types.UntypedRune:
            val = constant.ToInt(val)
        case types.UntypedFloat:
            val = constant.ToFloat(val)
        case types.UntypedComplex:
            val = constant.ToComplex(val)
        }
        name.SetVal(val)

        out.Append(ir.NewDecl(g.pos(decl), ir.ODCLCONST, name))
    }
}
```

```
func (g *irgen) funcDecl(out *ir.Nodes, decl *syntax.FuncDecl) {
    ... // Omitted code

    fn := ir.NewFunc(g.pos(decl))
    fn.Nname, _ = g.def(decl.Name)
    fn.Nname.Func = fn
    fn.Nname.Defn = fn

    ... // Omitted code

    if decl.Name.Value == "init" && decl.Recv == nil {
        g.target.Inits = append(g.target.Inits, fn)
    }

    saveHaveEmbed := g.haveEmbed
    saveCurDecl := g.curDecl
    g.curDecl = ""
    g.later(func() {
        defer func(b bool, s string) {
            g.haveEmbed = b
            g.curDecl = s
        }(g.haveEmbed, g.curDecl)

        g.haveEmbed = saveHaveEmbed
        g.curDecl = saveCurDecl
        if fn.Type().HasTParam() {
            g.topFuncIsGeneric = true
        }
        g.funcBody(fn, decl.Recv, decl.Type, decl.Body)
        g.topFuncIsGeneric = false
        if fn.Type().HasTParam() && fn.Body != nil {
            fn.Inl = &ir.Inline{
                Cost: 1,
                Dcl:  fn.Dcl,
                Body: fn.Body,
            }
        }
    })
    out.Append(fn)
}
```

# THE IR GENERATION FUNCTIONS

```
func (g *irgen) constDecl(out *ir.Nodes, decl *syntax.ConstDecl) {
    g.pragmaFlags(decl.Pragma, 0)

    for _, name := range decl.NameList {
        name, obj := g.def(name)

        // For untyped numeric constants, make sure the value
        // representation matches what the rest of the
        // compiler (really just iexport) expects.
        // TODO(mdempsky): Revisit after #43891 is resolved.
        val := obj.(*types2.Const).Val()
        switch name.Type() {
        case types.UntypedInt, types.UntypedRune:
            val = constant.ToInt(val)
        case types.UntypedFloat:
            val = constant.ToFloat(val)
        case types.UntypedComplex:
            val = constant.ToComplex(val)
        }
        name.SetVal(val)

        out.Append(ir.NewDecl(g.pos(decl), ir.ODCLCONST, name))
    }
}
```

```
func (g *irgen) funcDecl(out *ir.Nodes, decl *syntax.FuncDecl) {
    ... // Omitted code

    fn := ir.NewFunc(g.pos(decl))
    fn.Nname, _ = g.def(decl.Name)
    fn.Nname.Func = fn
    fn.Nname.Defn = fn

    ... // Omitted code

    if decl.Name.Value == "init" && decl.Recv == nil {
        g.target.Inits = append(g.target.Inits, fn)
    }

    saveHaveEmbed := g.haveEmbed
    saveCurDecl := g.curDecl
    g.curDecl = ""
    g.later(func() {
        defer func(b bool, s string) {
            g.haveEmbed = b
            g.curDecl = s
        }(g.haveEmbed, g.curDecl)

        g.haveEmbed = saveHaveEmbed
        g.curDecl = saveCurDecl
        if fn.Type().HasTParam() {
            g.topFuncIsGeneric = true
        }
        g.funcBody(fn, decl.Recv, decl.Type, decl.Body)
        g.topFuncIsGeneric = false
        if fn.Type().HasTParam() && fn.Body != nil {
            fn.Inl = &ir.Inline{
                Cost: 1,
                Dcl:  fn.Dcl,
                Body: fn.Body,
            }
        }
        out.Append(fn)
    })
}
```

# THE IR GENERATION FUNCTIONS

```
func (g *irgen) constDecl(out *ir.Nodes, decl *syntax.ConstDecl) {
    g.pragmaFlags(decl.Pragma, 0)

    for _, name := range decl.NameList {
        name, obj := g.def(name)

        // For untyped numeric constants, make sure the value
        // representation matches what the rest of the
        // compiler (really just iexport) expects.
        // TODO(mdempsky): Revisit after #43891 is resolved.
        val := obj.(*types2.Const).Val()
        switch name.Type() {
        case types.UntypedInt, types.UntypedRune:
            val = constant.ToInt(val)
        case types.UntypedFloat:
            val = constant.ToFloat(val)
        case types.UntypedComplex:
            val = constant.ToComplex(val)
        }
        name.SetVal(val)

        out.Append(ir.NewDecl(g.pos(decl), ir.ODCLCONST, name))
    }
}
```

```
func (g *irgen) funcDecl(out *ir.Nodes, decl *syntax.FuncDecl) {
    ... // Omitted code

    fn := ir.NewFunc(g.pos(decl))
    fn.Nname, _ = g.def(decl.Name)
    fn.Nname.Func = fn
    fn.Nname.Defn = fn

    ... // Omitted code

    if decl.Name.Value == "init" && decl.Recv == nil {
        g.target.Inits = append(g.target.Inits, fn)
    }

    saveHaveEmbed := g.haveEmbed
    saveCurDecl := g.curDecl
    g.curDecl = ""
    g.later(func() {
        defer func(b bool, s string) {
            g.haveEmbed = b
            g.curDecl = s
        }(g.haveEmbed, g.curDecl)

        g.haveEmbed = saveHaveEmbed
        g.curDecl = saveCurDecl
        if fn.Type().HasTParam() {
            g.topFuncIsGeneric = true
        }
        g.funcBody(fn, decl.Recv, decl.Type, decl.Body)
        g.topFuncIsGeneric = false
        if fn.Type().HasTParam() && fn.Body != nil {
            fn.Inl = &ir.Inline{
                Cost: 1,
                Dcl:  fn.Dcl,
                Body: fn.Body,
            }
        }
    })
    out.Append(fn)
}
```

# THE IR GENERATION FUNCTIONS

```
func (g *irgen) constDecl(out *ir.Nodes, decl *syntax.ConstDecl) {
    g.pragmaFlags(decl.Pragma, 0)

    for _, name := range decl.NameList {
        name, obj := g.def(name)

        // For untyped numeric constants, make sure the value
        // representation matches what the rest of the
        // compiler (really just iexport) expects.
        // TODO(mdempsky): Revisit after #43891 is resolved.
        val := obj.(*types2.Const).Val()
        switch name.Type() {
        case types.UntypedInt, types.UntypedRune:
            val = constant.ToInt(val)
        case types.UntypedFloat:
            val = constant.ToFloat(val)
        case types.UntypedComplex:
            val = constant.ToComplex(val)
        }
        name.SetVal(val)

        out.Append(ir.NewDecl(g.pos(decl), ir.ODCLCONST, name))
    }
}
```

```
func (g *irgen) funcDecl(out *ir.Nodes, decl *syntax.FuncDecl) {
    ... // Omitted code

    fn := ir.NewFunc(g.pos(decl))
    fn.Nname, _ = g.def(decl.Name)
    fn.Nname.Func = fn
    fn.Nname.Defn = fn

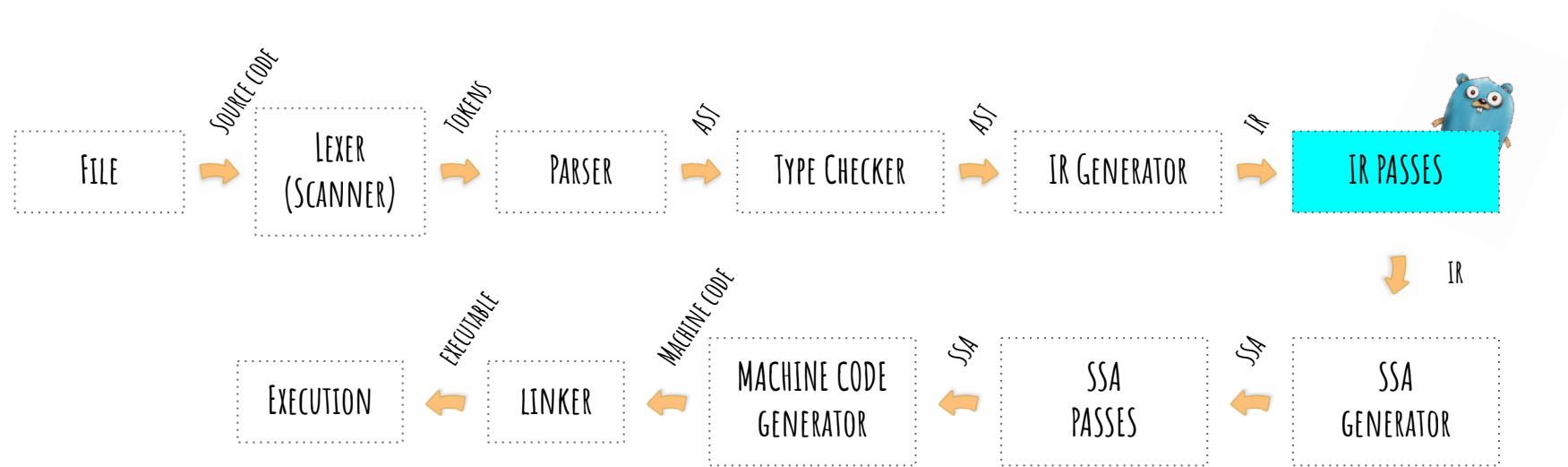
    ... // Omitted code

    if decl.Name.Value == "init" && decl.Recv == nil {
        g.target.Inits = append(g.target.Inits, fn)
    }

    saveHaveEmbed := g.haveEmbed
    saveCurDecl := g.curDecl
    g.curDecl = ""
    g.later(func() {
        defer func(b bool, s string) {
            g.haveEmbed = b
            g.curDecl = s
        }(g.haveEmbed, g.curDecl)

        g.haveEmbed = saveHaveEmbed
        g.curDecl = saveCurDecl
        if fn.Type().HasTParam() {
            g.topFuncIsGeneric = true
        }
        g.funcBody(fn, decl.Recv, decl.Type, decl.Body)
        g.topFuncIsGeneric = false
        if fn.Type().HasTParam() && fn.Body != nil {
            fn.Inl = &ir.Inline{
                Cost: 1,
                Dcl:  fn.Dcl,
                Body: fn.Body,
            }
        }
    })
    out.Append(fn)
}
```

# THE IR



# THE IR PASSES

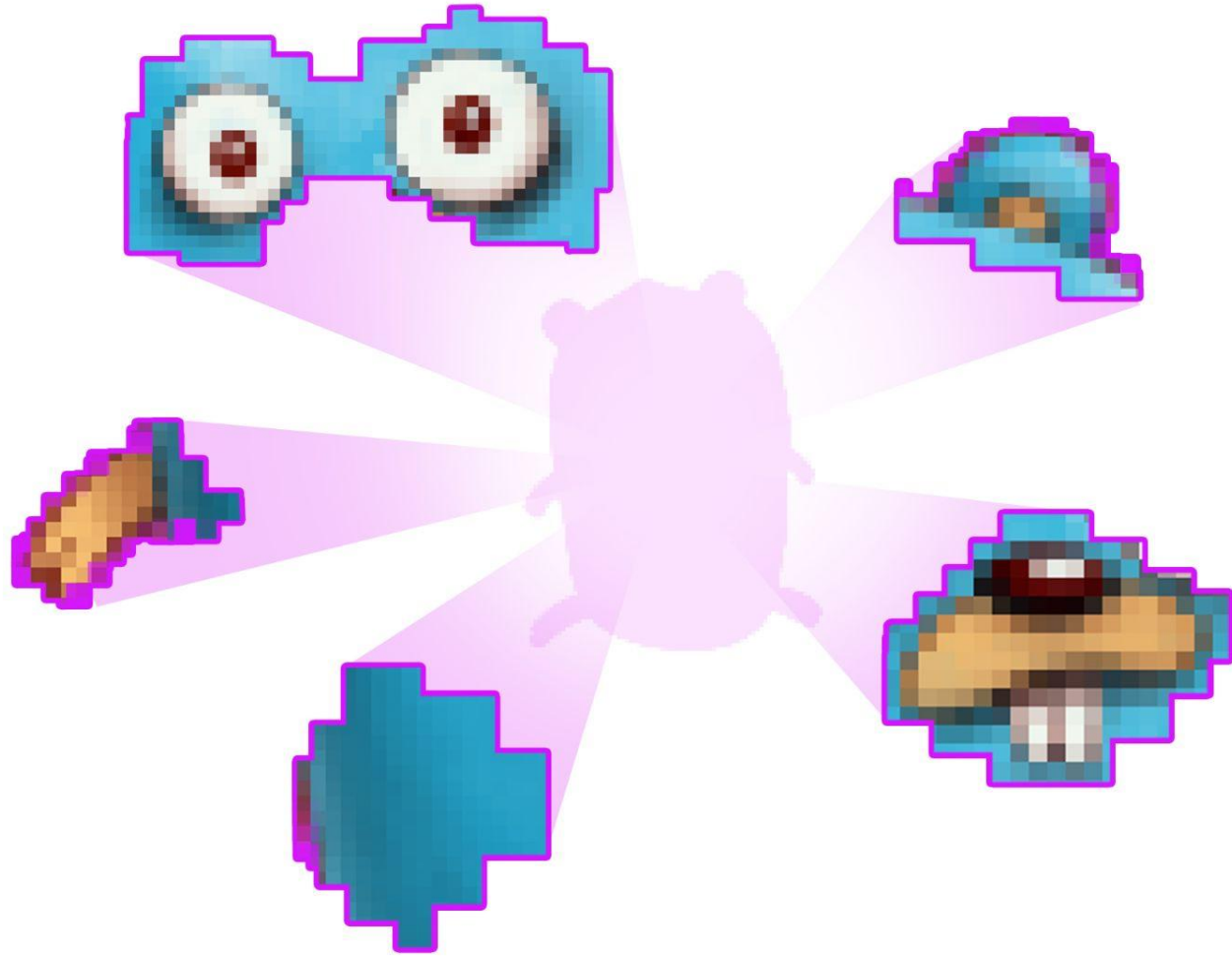
- Dead code elimination
- Function call inlining
- Devirtualize functions
- Escape analysis



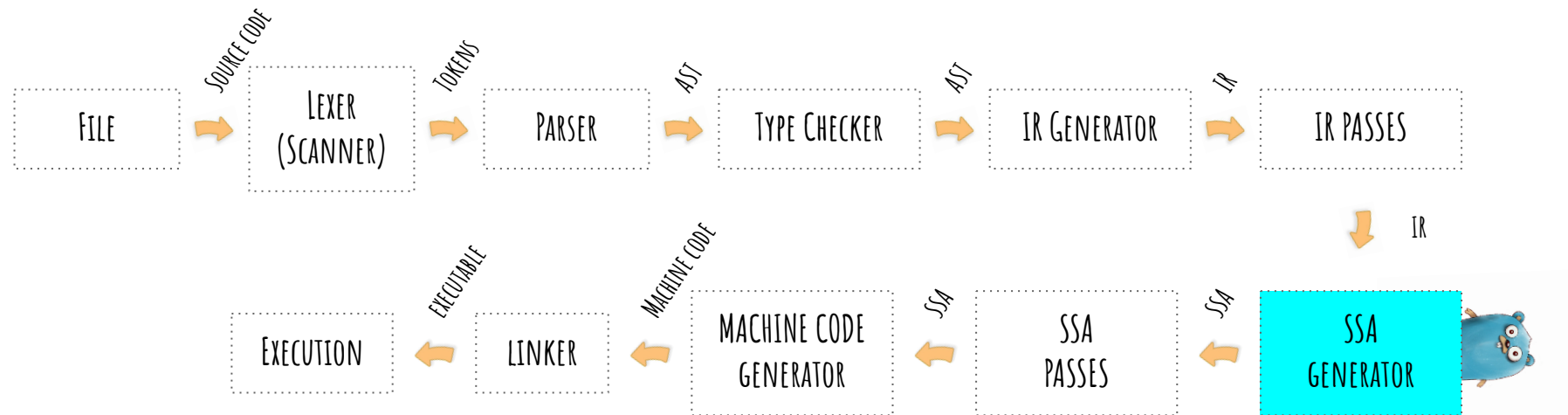
TOO MUCH INFO? TAKE A BREAK.  
LOOK... HERE IS A KITTEN



# SSA (STATIC SINGLE ASSIGNMENT)



# SSA (STATIC SINGLE ASSIGNMENT)

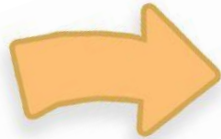


# SSA (STATIC SINGLE ASSIGNMENT)

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



```
b1:  
v1 (?) = InitMem <mem>  
v2 (?) = SP <uintptr>  
v3 (?) = SB <uintptr>  
v4 (?) = ConstInterface <any>  
v5 (?) = ArrayMake1 <[1]any> v4  
v6 (6) = VarDef <mem> {.autotmp_8} v1  
v7 (6) = LocalAddr <*[1]any> {.autotmp_8} v2 v6  
v8 (6) = Store <mem> {[1]any} v7 v5 v6  
v9 (6) = LocalAddr <*[1]any> {.autotmp_8} v2 v8  
v10 (?) = Addr <uint8> {type.string} v3  
v11 (?) = Addr <string> {main..stmp_0} v3  
v12 (6) = IMake <any> v10 v11  
v13 (6) = NilCheck <void> v9 v8  
v14 (?) = Const64 <int> [0] (fmt.n[int], fmt..autotmp_0[int])  
v15 (?) = Const64 <int> [1]  
v16 (6) = PtrIndex <*any> v9 v14  
v17 (6) = Store <mem> {any} v16 v12 v8  
v18 (6) = NilCheck <void> v9 v17  
v19 (6) = Copy <*any> v9  
v20 (6) = IsSliceInBounds <bool> v14 v15  
v25 (?) = ConstInterface <error> (fmt.err[error], fmt..autotmp_1[error])  
v28 (?) = Addr <uint8> {go.itab.*os.File,io.Writer} v3  
v29 (?) = Addr <**os.File> {os.Stdout} v3  
If v20 → b2 b3 (likely) (6)  
  
b2: ← b1  
v23 (6) = Sub64 <int> v15 v14  
v24 (6) = SliceMake <[ ]any> v19 v23 v23 (fmt.a[[ ]any])  
v26 (6) = Copy <mem> v17  
v27 (+6) = InlMark <void> [0] v26  
v30 (294) = Load <*os.File> v29 v26  
v31 (294) = IMake <io.Writer> v28 v30  
v32 (294) = StaticLECall <int,error,mem> {AuxCall{fmt.Fprintln}} [40] v31 v24 v26  
v33 (294) = SelectN <mem> [2] v32  
v34 (294) = SelectN <int> [0] v32  
v35 (294) = SelectN <int> [0] v32 (fmt.n[int], fmt..autotmp_0[int])  
v36 (294) = SelectN <error> [1] v32 (fmt.err[error], fmt..autotmp_1[error])  
Plain → b4 (+6)  
  
b3: ← b1  
  
v21 (6) = Copy <mem> v17  
v22 (6) = PanicBounds <mem> [6] v14 v15 v21  
Exit v22 (6)  
  
b4: ← b2  
  
v38 (7) = Copy <mem> v33  
v37 (7) = MakeResult <mem> v38  
Ret v37 (7)  
  
name fmt.a[[ ]any]: v24  
name fmt.n[int]: v14 v35  
name fmt.err[error]: v25 v36  
name fmt..autotmp_0[int]: v14 v35  
name fmt..autotmp_1[error]: v25 v36
```

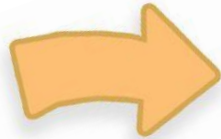
Generated with GOSSAFUNC=main.main go build hello.go

# SSA (STATIC SINGLE ASSIGNMENT)

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



```
b1:  
v1 (?) = InitMem <mem>  
v2 (?) = SP <uintptr>  
v3 (?) = SB <uintptr>  
v4 (?) = ConstInterface <any>  
v5 (?) = ArrayMakel <[1]any> v4  
v6 (6) = VarDef <mem> {.autotmp_8} v1  
v7 (6) = LocalAddr <*[1]any> {.autotmp_8} v2 v6  
v8 (6) = Store <mem> {[1]any} v7 v5 v6  
v9 (6) = LocalAddr <*[1]any> {.autotmp_8} v2 v8  
v10 (?) = Addr <uint8> {type.string} v3  
v11 (?) = Addr <string> {main..stmp_0} v3  
v12 (6) = IMake <any> v10 v11  
v13 (6) = NilCheck <void> v9 v8  
v14 (?) = Const64 <int> [0] (fmt.n[int], fmt..autotmp_0[int])  
v15 (?) = Const64 <int> [1]  
v16 (6) = PtrIndex <any> v9 v14  
v17 (6) = Store <mem> {any} v16 v12 v8  
v18 (6) = NilCheck <void> v9 v17  
v19 (6) = Copy <any> v9  
v20 (6) = IsSliceInBounds <bool> v14 v15  
v25 (?) = ConstInterface <error> (fmt.err[error], fmt..autotmp_1[error])  
v28 (?) = Addr <uint8> {go.itab.*os.File,io.Writer} v3  
v29 (?) = Addr <*os.File> {os.Stdout} v3  
If v20 → b2 b3 (likely) (6)
```

```
b2: ← b1  
v23 (6) = Sub64 <int> v15 v14  
v24 (6) = SliceMake <[1]any> v19 v23 v23 (fmt.a[[1]any])  
v26 (6) = Copy <mem> v17  
v27 (+6) = InlMark <void> [0] v26  
v30 (294) = Load <*os.File> v29 v26  
v31 (294) = IMake <io.Writer> v28 v30  
v32 (294) = StaticLECall <int,error,mem> {AuxCall{fmt.Fprintln}} [40] v31 v24 v26  
v33 (294) = SelectN <mem> [2] v32  
v34 (294) = SelectN <int> [0] v32  
v35 (294) = SelectN <int> [0] v32 (fmt.n[int], fmt..autotmp_0[int])  
v36 (294) = SelectN <error> [1] v32 (fmt.err[error], fmt..autotmp_1[error])  
Plain → b4 (+6)
```

```
b3: ← b1  
v21 (6) = Copy <mem> v17  
v22 (6) = PanicBounds <mem> [6] v14 v15 v21  
Exit v22 (6)
```

```
b4: ← b2  
v38 (7) = Copy <mem> v33  
v37 (7) = MakeResult <mem> v38  
Ret v37 (7)
```

```
name fmt.a[[1]any]: v24  
name fmt.n[int]: v14 v35  
name fmt.err[error]: v25 v36  
name fmt..autotmp_0[int]: v14 v35  
name fmt..autotmp_1[error]: v25 v36
```

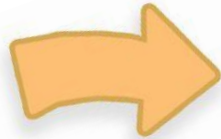
Generated with GOSSAFUNC=main.main go build hello.go

# SSA (STATIC SINGLE ASSIGNMENT)

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



b1:

```
v1 (?) = InitMem <mem>  
v2 (?) = SP <uintptr>  
v3 (?) = SB <uintptr>  
v4 (?) = ConstInterface <any>  
v5 (?) = ArrayMake1 <[1]any> v4  
v6 (6) = VarDef <mem> {.autotmp_8} v1  
v7 (6) = LocalAddr <*[1]any> {.autotmp_8} v2 v6  
v8 (6) = Store <mem> {[1]any} v7 v5 v6  
v9 (6) = LocalAddr <*[1]any> {.autotmp_8} v2 v8  
v10 (?) = Addr <*uint8> {type.string} v3  
v11 (?) = Addr <*string> {main..stmp_0} v3  
v12 (6) = IMake <any> v10 v11  
v13 (6) = NilCheck <void> v9 v8  
v14 (?) = Const64 <int> [0] (fmt.n[int], fmt..autotmp_0[int])  
v15 (?) = Const64 <int> [1]  
v16 (6) = PtrIndex <*any> v9 v14  
v17 (6) = Store <mem> {any} v16 v12 v8  
v18 (6) = NilCheck <void> v9 v17  
v19 (6) = Copy <*any> v9  
v20 (6) = IsSliceInBounds <bool> v14 v15  
v25 (?) = ConstInterface <error> (fmt.err[error], fmt..autotmp_1[error])  
v28 (?) = Addr <*uint8> {go.itab.*os.File,io.Writer} v3  
v29 (?) = Addr <**os.File> {os.Stdout} v3  
If v20 → b2 b3 (likely) (6)
```

b2: ← b1

```
v23 (6) = Sub64 <int> v15 v14  
v24 (6) = SliceMake <[ ]any> v19 v23 v23 (fmt.a[[ ]any])  
v26 (6) = Copy <mem> v17  
v27 (+6) = InlMark <void> [0] v26  
v30 (294) = Load <*os.File> v29 v26  
v31 (294) = IMake <io.Writer> v28 v30  
v32 (294) = StaticLECall <int,error,mem> {AuxCall{fmt.Fprintln}} [40] v31 v24 v26  
v33 (294) = SelectN <mem> [2] v32  
v34 (294) = SelectN <int> [0] v32  
v35 (294) = SelectN <int> [0] v32 (fmt.n[int], fmt..autotmp_0[int])  
v36 (294) = SelectN <error> [1] v32 (fmt.err[error], fmt..autotmp_1[error])  
Plain → b4 (+6)
```

b3: ← b1

```
v21 (6) = Copy <mem> v17  
v22 (6) = PanicBounds <mem> [6] v14 v15 v21  
Exit v22 (6)
```

b4: ← b2

```
v38 (7) = Copy <mem> v33  
v37 (7) = MakeResult <mem> v38  
Ret v37 (7)
```

```
name fmt.a[[ ]any]: v24  
name fmt.n[int]: v14 v35  
name fmt.err[error]: v25 v36  
name fmt..autotmp_0[int]: v14 v35  
name fmt..autotmp_1[error]: v25 v36
```

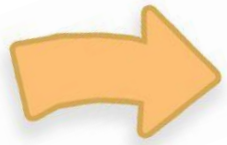
Generated with GOSSAFUNC=main.main go build hello.go

# SSA (STATIC SINGLE ASSIGNMENT)

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



```
# Name: hello.init  
# Package: hello  
# Synthetic: package initializer  
func init():
```

```
0:  
    t0 = *init$guard  
    if t0 goto 2 else 1  
1:  
    *init$guard = true:bool  
    t1 = fmt.init()  
    jump 2  
2:  
    return
```

```
# Name: hello.main  
# Package: hello  
# Location: hello.go:8:6  
func main():
```

```
0:  
    t0 = new [1]any (varargs)  
    t1 = &t0[0:int]  
    t2 = make any <- string ("hello world!":string)  
    *t1 = t2  
    t3 = slice t0[:]  
    t4 = fmt.Println(t3...)  
    return
```

```
entry:0 S:2  
    bool
```

```
init$start P:1 S:1  
    ()
```

```
init$done P:2 S:0
```

```
entry:0 S:0  
    *[1]any  
    *any  
    any
```

```
    []any  
(n int, err error)
```

# SSA (STATIC SINGLE ASSIGNMENT)

*// stmtList converts the statement list n to SSA and adds it to s.*

```
func (s *state) stmtList(l ir.Nodes) {  
    for _, n := range l {  
        s.stmt(n)  
    }  
}
```

*// stmt converts the statement n to SSA and adds it to s.*

```
func (s *state) stmt(n ir.Node) {
```

```
case ir.ODCL:  
    n := n.(*ir.Decl)  
    if v := n.X; v.Esc() == ir.EscHeap {  
        s.newHeapaddr(v)  
    }  
  
case ir.OVARDEF:  
    n := n.(*ir.UnaryExpr)  
    if !s.canSSA(n.X) {  
        s.vars[memVar] = s.newValue1Apos(ssa.OpVarDef, types.TypeMem, n.X.(*ir.Name), s.mem(), false)  
    }  
}
```

```
case ir.OIF:  
    n := n.(*ir.IfStmt)  
    if ir.IsConst(n.Cond, constant.Bool) {  
        s.stmtList(n.Cond.Init())  
        if ir.BoolVal(n.Cond) {  
            s.stmtList(n.Body)  
        } else {  
            s.stmtList(n.Else)  
        }  
        break  
    }  
  
bEnd := s.f.NewBlock(ssa.BlockPlain)  
var likely int8  
if n.Likely {  
    likely = 1  
}  
var bThen *ssa.Block  
if len(n.Body) != 0 {  
    bThen = s.f.NewBlock(ssa.BlockPlain)  
} else {  
    bThen = bEnd  
}  
var bElse *ssa.Block  
if len(n.Else) != 0 {  
    bElse = s.f.NewBlock(ssa.BlockPlain)  
} else {  
    bElse = bEnd  
}  
s.condBranch(n.Cond, bThen, bElse, likely)  
  
if len(n.Body) != 0 {  
    s.startBlock(bThen)  
    s.stmtList(n.Body)  
    if b := s.endBlock(); b != nil {  
        b.AddEdgeTo(bEnd)  
    }  
}  
if len(n.Else) != 0 {  
    s.startBlock(bElse)  
    s.stmtList(n.Else)  
    if b := s.endBlock(); b != nil {  
        b.AddEdgeTo(bEnd)  
    }  
}  
s.startBlock(bEnd)
```



# SSA (STATIC SINGLE ASSIGNMENT)

*// stmtList converts the statement list n to SSA and adds it to s.*

```
func (s *state) stmtList(l ir.Nodes) {  
    for _, n := range l {  
        s.stmt(n)  
    }  
}
```

*// stmt converts the statement n to SSA and adds it to s.*

```
func (s *state) stmt(n ir.Node) {
```

```
case ir.ODCL:  
    n := n.(*ir.Decl)  
    if v := n.X; v.Esc() == ir.EscHeap {  
        s.newHeapaddr(v)  
    }  
  
case ir.OVARDEF:  
    n := n.(*ir.UnaryExpr)  
    if !s.canSSA(n.X) {  
        s.vars[memVar] = s.newValue1Apos(ssa.OpVarDef, types.TypeMem, n.X.(*ir.Name), s.mem(), false)  
    }  
}
```

```
case ir.OIF:  
    n := n.(*ir.IfStmt)  
    if ir.IsConst(n.Cond, constant.Bool) {  
        s.stmtList(n.Cond.Init())  
        if ir.BoolVal(n.Cond) {  
            s.stmtList(n.Body)  
        } else {  
            s.stmtList(n.Else)  
        }  
        break  
    }  
  
bEnd := s.f.NewBlock(ssa.BlockPlain)  
var likely int8  
if n.Likely {  
    likely = 1  
}  
var bThen *ssa.Block  
if len(n.Body) != 0 {  
    bThen = s.f.NewBlock(ssa.BlockPlain)  
} else {  
    bThen = bEnd  
}  
var bElse *ssa.Block  
if len(n.Else) != 0 {  
    bElse = s.f.NewBlock(ssa.BlockPlain)  
} else {  
    bElse = bEnd  
}  
s.condBranch(n.Cond, bThen, bElse, likely)  
  
if len(n.Body) != 0 {  
    s.startBlock(bThen)  
    s.stmtList(n.Body)  
    if b := s.endBlock(); b != nil {  
        b.AddEdgeTo(bEnd)  
    }  
}  
if len(n.Else) != 0 {  
    s.startBlock(bElse)  
    s.stmtList(n.Else)  
    if b := s.endBlock(); b != nil {  
        b.AddEdgeTo(bEnd)  
    }  
}  
s.startBlock(bEnd)
```

# SSA (STATIC SINGLE ASSIGNMENT)

*// stmtList converts the statement list n to SSA and adds it to s.*

```
func (s *state) stmtList(l ir.Nodes) {  
    for _, n := range l {  
        s.stmt(n)  
    }  
}
```

*// stmt converts the statement n to SSA and adds it to s.*

```
func (s *state) stmt(n ir.Node) {
```

```
case ir.ODCL:  
    n := n.(*ir.Decl)  
    if v := n.X; v.Esc() == ir.EscHeap {  
        s.newHeapaddr(v)  
    }
```

```
case ir.OVARDEF:  
    n := n.(*ir.UnaryExpr)  
    if !s.canSSA(n.X) {  
        s.vars[memVar] = s.newValue1Apos(ssa.OpVarDef, types.TypeMem, n.X.(*ir.Name), s.mem(), false)  
    }
```

```
case ir.OIF:  
    n := n.(*ir.IfStmt)  
    if ir.IsConst(n.Cond, constant.Bool) {  
        s.stmtList(n.Cond.Init())  
        if ir.BoolVal(n.Cond) {  
            s.stmtList(n.Body)  
        } else {  
            s.stmtList(n.Else)  
        }  
        break  
    }  
  
    bEnd := s.f.NewBlock(ssa.BlockPlain)  
    var likely int8  
    if n.Likely {  
        likely = 1  
    }  
    var bThen *ssa.Block  
    if len(n.Body) != 0 {  
        bThen = s.f.NewBlock(ssa.BlockPlain)  
    } else {  
        bThen = bEnd  
    }  
    var bElse *ssa.Block  
    if len(n.Else) != 0 {  
        bElse = s.f.NewBlock(ssa.BlockPlain)  
    } else {  
        bElse = bEnd  
    }  
    s.condBranch(n.Cond, bThen, bElse, likely)  
  
    if len(n.Body) != 0 {  
        s.startBlock(bThen)  
        s.stmtList(n.Body)  
        if b := s.endBlock(); b != nil {  
            b.AddEdgeTo(bEnd)  
        }  
    }  
    if len(n.Else) != 0 {  
        s.startBlock(bElse)  
        s.stmtList(n.Else)  
        if b := s.endBlock(); b != nil {  
            b.AddEdgeTo(bEnd)  
        }  
    }  
    s.startBlock(bEnd)
```

# SSA (STATIC SINGLE ASSIGNMENT)

*// stmtList converts the statement list n to SSA and adds it to s.*

```
func (s *state) stmtList(l ir.Nodes) {  
    for _, n := range l {  
        s.stmt(n)  
    }  
}
```

*// stmt converts the statement n to SSA and adds it to s.*

```
func (s *state) stmt(n ir.Node) {
```

```
    case ir.ODCL:  
        n := n.(*ir.Decl)  
        if v := n.X; v.Esc() == ir.EscHeap {  
            s.newHeapaddr(v)  
        }  
}
```

```
    case ir.OVARDEF:  
        n := n.(*ir.UnaryExpr)  
        if !s.canSSA(n.X) {  
            s.vars[memVar] = s.newValue1Apos(ssa.OpVarDef, types.TypeMem, n.X.(*ir.Name), s.mem(), false)  
        }  
}
```

```
    case ir.OIF:  
        n := n.(*ir.IfStmt)  
        if ir.IsConst(n.Cond, constant.Bool) {  
            s.stmtList(n.Cond.Init())  
            if ir.BoolVal(n.Cond) {  
                s.stmtList(n.Body)  
            } else {  
                s.stmtList(n.Else)  
            }  
            break  
        }  
  
        bEnd := s.f.NewBlock(ssa.BlockPlain)  
        var likely int8  
        if n.Likely {  
            likely = 1  
        }  
        var bThen *ssa.Block  
        if len(n.Body) != 0 {  
            bThen = s.f.NewBlock(ssa.BlockPlain)  
        } else {  
            bThen = bEnd  
        }  
        var bElse *ssa.Block  
        if len(n.Else) != 0 {  
            bElse = s.f.NewBlock(ssa.BlockPlain)  
        } else {  
            bElse = bEnd  
        }  
        s.condBranch(n.Cond, bThen, bElse, likely)  
  
        if len(n.Body) != 0 {  
            s.startBlock(bThen)  
            s.stmtList(n.Body)  
            if b := s.endBlock(); b != nil {  
                b.AddEdgeTo(bEnd)  
            }  
        }  
        if len(n.Else) != 0 {  
            s.startBlock(bElse)  
            s.stmtList(n.Else)  
            if b := s.endBlock(); b != nil {  
                b.AddEdgeTo(bEnd)  
            }  
        }  
        s.startBlock(bEnd)
```

# SSA (STATIC SINGLE ASSIGNMENT)

*// stmtList converts the statement list n to SSA and adds it to s.*

```
func (s *state) stmtList(l ir.Nodes) {  
    for _, n := range l {  
        s.stmt(n)  
    }  
}
```

*// stmt converts the statement n to SSA and adds it to s.*

```
func (s *state) stmt(n ir.Node) {
```

```
case ir.ODCL:  
    n := n.(*ir.Decl)  
    if v := n.X; v.Esc() == ir.EscHeap {  
        s.newHeapaddr(v)  
    }
```

```
case ir.OVARDEF:  
    n := n.(*ir.UnaryExpr)  
    if !s.canSSA(n.X) {  
        s.vars[memVar] = s.newValue1Apos(ssa.OpVarDef, types.TypeMem, n.X.(*ir.Name), s.mem(), false)  
    }
```

```
case ir.OIF:  
    n := n.(*ir.IfStmt)  
    if ir.IsConst(n.Cond, constant.Bool) {  
        s.stmtList(n.Cond.Init())  
        if ir.BoolVal(n.Cond) {  
            s.stmtList(n.Body)  
        } else {  
            s.stmtList(n.Else)  
        }  
        break  
    }  
  
    bEnd := s.f.NewBlock(ssa.BlockPlain)  
    var likely int8  
    if n.Likely {  
        likely = 1  
    }  
    var bThen *ssa.Block  
    if len(n.Body) != 0 {  
        bThen = s.f.NewBlock(ssa.BlockPlain)  
    } else {  
        bThen = bEnd  
    }  
    var bElse *ssa.Block  
    if len(n.Else) != 0 {  
        bElse = s.f.NewBlock(ssa.BlockPlain)  
    } else {  
        bElse = bEnd  
    }  
    s.condBranch(n.Cond, bThen, bElse, likely)  
  
    if len(n.Body) != 0 {  
        s.startBlock(bThen)  
        s.stmtList(n.Body)  
        if b := s.endBlock(); b != nil {  
            b.AddEdgeTo(bEnd)  
        }  
    }  
    if len(n.Else) != 0 {  
        s.startBlock(bElse)  
        s.stmtList(n.Else)  
        if b := s.endBlock(); b != nil {  
            b.AddEdgeTo(bEnd)  
        }  
    }  
    s.startBlock(bEnd)
```

# SSA (STATIC SINGLE ASSIGNMENT)

*// stmtList converts the statement list n to SSA and adds it to s.*

```
func (s *state) stmtList(l ir.Nodes) {  
    for _, n := range l {  
        s.stmt(n)  
    }  
}
```

*// stmt converts the statement n to SSA and adds it to s.*

```
func (s *state) stmt(n ir.Node) {
```

```
case ir.ODCL:  
    n := n.(*ir.Decl)  
    if v := n.X; v.Esc() == ir.EscHeap {  
        s.newHeapaddr(v)  
    }  
  
case ir.OVARDEF:  
    n := n.(*ir.UnaryExpr)  
    if !s.canSSA(n.X) {  
        s.vars[memVar] = s.newValue1Apos(ssa.OpVarDef, types.TypeMem, n.X.(*ir.Name), s.mem(), false)  
    }  
}
```

```
case ir.OIF:  
    n := n.(*ir.IfStmt)  
    if ir.IsConst(n.Cond, constant.Bool) {  
        s.stmtList(n.Cond.Init())  
        if ir.BoolVal(n.Cond) {  
            s.stmtList(n.Body)  
        } else {  
            s.stmtList(n.Else)  
        }  
        break  
    }  
  
    bEnd := s.f.NewBlock(ssa.BlockPlain)  
    var likely int8  
    if n.Likely {  
        likely = 1  
    }  
    var bThen *ssa.Block  
    if len(n.Body) != 0 {  
        bThen = s.f.NewBlock(ssa.BlockPlain)  
    } else {  
        bThen = bEnd  
    }  
    var bElse *ssa.Block  
    if len(n.Else) != 0 {  
        bElse = s.f.NewBlock(ssa.BlockPlain)  
    } else {  
        bElse = bEnd  
    }  
    s.condBranch(n.Cond, bThen, bElse, likely)  
  
    if len(n.Body) != 0 {  
        s.startBlock(bThen)  
        s.stmtList(n.Body)  
        if b := s.endBlock(); b != nil {  
            b.AddEdgeTo(bEnd)  
        }  
    }  
    if len(n.Else) != 0 {  
        s.startBlock(bElse)  
        s.stmtList(n.Else)  
        if b := s.endBlock(); b != nil {  
            b.AddEdgeTo(bEnd)  
        }  
    }  
    s.startBlock(bEnd)
```

# SSA (STATIC SINGLE ASSIGNMENT)

*// stmtList converts the statement list n to SSA and adds it to s.*

```
func (s *state) stmtList(l ir.Nodes) {  
    for _, n := range l {  
        s.stmt(n)  
    }  
}
```

*// stmt converts the statement n to SSA and adds it to s.*

```
func (s *state) stmt(n ir.Node) {
```

```
case ir.ODCL:  
    n := n.(*ir.Decl)  
    if v := n.X; v.Esc() == ir.EscHeap {  
        s.newHeapaddr(v)  
    }  
  
case ir.OVARDEF:  
    n := n.(*ir.UnaryExpr)  
    if !s.canSSA(n.X) {  
        s.vars[memVar] = s.newValue1Apos(ssa.OpVarDef, types.TypeMem, n.X.(*ir.Name), s.mem(), false)  
    }  
}
```

```
case ir.OIF:  
    n := n.(*ir.IfStmt)  
    if ir.IsConst(n.Cond, constant.Bool) {  
        s.stmtList(n.Cond.Init())  
        if ir.BoolVal(n.Cond) {  
            s.stmtList(n.Body)  
        } else {  
            s.stmtList(n.Else)  
        }  
    }  
    break  
}  
  
bEnd := s.f.NewBlock(ssa.BlockPlain)  
var likely int8  
if n.Likely {  
    likely = 1  
}  
var bThen *ssa.Block  
if len(n.Body) != 0 {  
    bThen = s.f.NewBlock(ssa.BlockPlain)  
} else {  
    bThen = bEnd  
}  
var bElse *ssa.Block  
if len(n.Else) != 0 {  
    bElse = s.f.NewBlock(ssa.BlockPlain)  
} else {  
    bElse = bEnd  
}  
s.condBranch(n.Cond, bThen, bElse, likely)  
  
if len(n.Body) != 0 {  
    s.startBlock(bThen)  
    s.stmtList(n.Body)  
    if b := s.endBlock(); b != nil {  
        b.AddEdgeTo(bEnd)  
    }  
}  
if len(n.Else) != 0 {  
    s.startBlock(bElse)  
    s.stmtList(n.Else)  
    if b := s.endBlock(); b != nil {  
        b.AddEdgeTo(bEnd)  
    }  
}  
s.startBlock(bEnd)
```

# SSA (STATIC SINGLE ASSIGNMENT)

*// stmtList converts the statement list n to SSA and adds it to s.*

```
func (s *state) stmtList(l ir.Nodes) {  
    for _, n := range l {  
        s.stmt(n)  
    }  
}
```

*// stmt converts the statement n to SSA and adds it to s.*

```
func (s *state) stmt(n ir.Node) {
```

```
case ir.ODCL:  
    n := n.(*ir.Decl)  
    if v := n.X; v.Esc() == ir.EscHeap {  
        s.newHeapaddr(v)  
    }  
  
case ir.OVARDEF:  
    n := n.(*ir.UnaryExpr)  
    if !s.canSSA(n.X) {  
        s.vars[memVar] = s.newValue1Apos(ssa.OpVarDef, types.TypeMem, n.X.(*ir.Name), s.mem(), false)  
    }  
}
```

```
case ir.OIF:  
    n := n.(*ir.IfStmt)  
    if ir.IsConst(n.Cond, constant.Bool) {  
        s.stmtList(n.Cond.Init())  
        if ir.BoolVal(n.Cond) {  
            s.stmtList(n.Body)  
        } else {  
            s.stmtList(n.Else)  
        }  
        break  
    }  
  
bEnd := s.f.NewBlock(ssa.BlockPlain)  
var likely int8  
if n.Likely {  
    likely = 1  
}  
var bThen *ssa.Block  
if len(n.Body) != 0 {  
    bThen = s.f.NewBlock(ssa.BlockPlain)  
} else {  
    bThen = bEnd  
}  
var bElse *ssa.Block  
if len(n.Else) != 0 {  
    bElse = s.f.NewBlock(ssa.BlockPlain)  
} else {  
    bElse = bEnd  
}  
s.condBranch(n.Cond, bThen, bElse, likely)  
  
if len(n.Body) != 0 {  
    s.startBlock(bThen)  
    s.stmtList(n.Body)  
    if b := s.endBlock(); b != nil {  
        b.AddEdgeTo(bEnd)  
    }  
}  
if len(n.Else) != 0 {  
    s.startBlock(bElse)  
    s.stmtList(n.Else)  
    if b := s.endBlock(); b != nil {  
        b.AddEdgeTo(bEnd)  
    }  
}  
s.startBlock(bEnd)
```

# SSA (STATIC SINGLE ASSIGNMENT)

*// stmtList converts the statement list n to SSA and adds it to s.*

```
func (s *state) stmtList(l ir.Nodes) {  
    for _, n := range l {  
        s.stmt(n)  
    }  
}
```

*// stmt converts the statement n to SSA and adds it to s.*

```
func (s *state) stmt(n ir.Node) {
```

```
case ir.ODCL:  
    n := n.(*ir.Decl)  
    if v := n.X; v.Esc() == ir.EscHeap {  
        s.newHeapaddr(v)  
    }
```

```
case ir.OVARDEF:  
    n := n.(*ir.UnaryExpr)  
    if !s.canSSA(n.X) {  
        s.vars[memVar] = s.newValue1Apos(ssa.OpVarDef, types.TypeMem, n.X.(*ir.Name), s.mem(), false)  
    }
```

```
case ir.OIF:  
    n := n.(*ir.IfStmt)  
    if ir.IsConst(n.Cond, constant.Bool) {  
        s.stmtList(n.Cond.Init())  
        if ir.BoolVal(n.Cond) {  
            s.stmtList(n.Body)  
        } else {  
            s.stmtList(n.Else)  
        }  
        break  
    }  
  
    bEnd := s.f.NewBlock(ssa.BlockPlain)  
    var likely int8  
    if n.Likely {  
        likely = 1  
    }  
    var bThen *ssa.Block  
    if len(n.Body) != 0 {  
        bThen = s.f.NewBlock(ssa.BlockPlain)  
    } else {  
        bThen = bEnd  
    }  
    var bElse *ssa.Block  
    if len(n.Else) != 0 {  
        bElse = s.f.NewBlock(ssa.BlockPlain)  
    } else {  
        bElse = bEnd  
    }  
    s.condBranch(n.Cond, bThen, bElse, likely)  
  
    if len(n.Body) != 0 {  
        s.startBlock(bThen)  
        s.stmtList(n.Body)  
        if b := s.endBlock(); b != nil {  
            b.AddEdgeTo(bEnd)  
        }  
    }  
    if len(n.Else) != 0 {  
        s.startBlock(bElse)  
        s.stmtList(n.Else)  
        if b := s.endBlock(); b != nil {  
            b.AddEdgeTo(bEnd)  
        }  
    }  
    s.startBlock(bEnd)
```



# SSA (STATIC SINGLE ASSIGNMENT)

*// stmtList converts the statement list n to SSA and adds it to s.*

```
func (s *state) stmtList(l ir.Nodes) {  
    for _, n := range l {  
        s.stmt(n)  
    }  
}
```

*// stmt converts the statement n to SSA and adds it to s.*

```
func (s *state) stmt(n ir.Node) {
```

```
case ir.ODCL:  
    n := n.(*ir.Decl)  
    if v := n.X; v.Esc() == ir.EscHeap {  
        s.newHeapaddr(v)  
    }
```

```
case ir.OVARDEF:  
    n := n.(*ir.UnaryExpr)  
    if !s.canSSA(n.X) {  
        s.vars[memVar] = s.newValue1Apos(ssa.OpVarDef, types.TypeMem, n.X.(*ir.Name), s.mem(), false)  
    }
```

```
case ir.OIF:  
    n := n.(*ir.IfStmt)  
    if ir.IsConst(n.Cond, constant.Bool) {  
        s.stmtList(n.Cond.Init())  
        if ir.BoolVal(n.Cond) {  
            s.stmtList(n.Body)  
        } else {  
            s.stmtList(n.Else)  
        }  
        break  
    }  
  
    bEnd := s.f.NewBlock(ssa.BlockPlain)  
    var likely int8  
    if n.Likely {  
        likely = 1  
    }  
    var bThen *ssa.Block  
    if len(n.Body) != 0 {  
        bThen = s.f.NewBlock(ssa.BlockPlain)  
    } else {  
        bThen = bEnd  
    }  
    var bElse *ssa.Block  
    if len(n.Else) != 0 {  
        bElse = s.f.NewBlock(ssa.BlockPlain)  
    } else {  
        bElse = bEnd  
    }  
    s.condBranch(n.Cond, bThen, bElse, likely)  
  
    if len(n.Body) != 0 {  
        s.startBlock(bThen)  
        s.stmtList(n.Body)  
        if b := s.endBlock(); b != nil {  
            b.AddEdgeTo(bEnd)  
        }  
    }  
    if len(n.Else) != 0 {  
        s.startBlock(bElse)  
        s.stmtList(n.Else)  
        if b := s.endBlock(); b != nil {  
            b.AddEdgeTo(bEnd)  
        }  
    }  
    s.startBlock(bEnd)
```

# SSA (STATIC SINGLE ASSIGNMENT)

*// stmtList converts the statement list n to SSA and adds it to s.*

```
func (s *state) stmtList(l ir.Nodes) {  
    for _, n := range l {  
        s.stmt(n)  
    }  
}
```

*// stmt converts the statement n to SSA and adds it to s.*

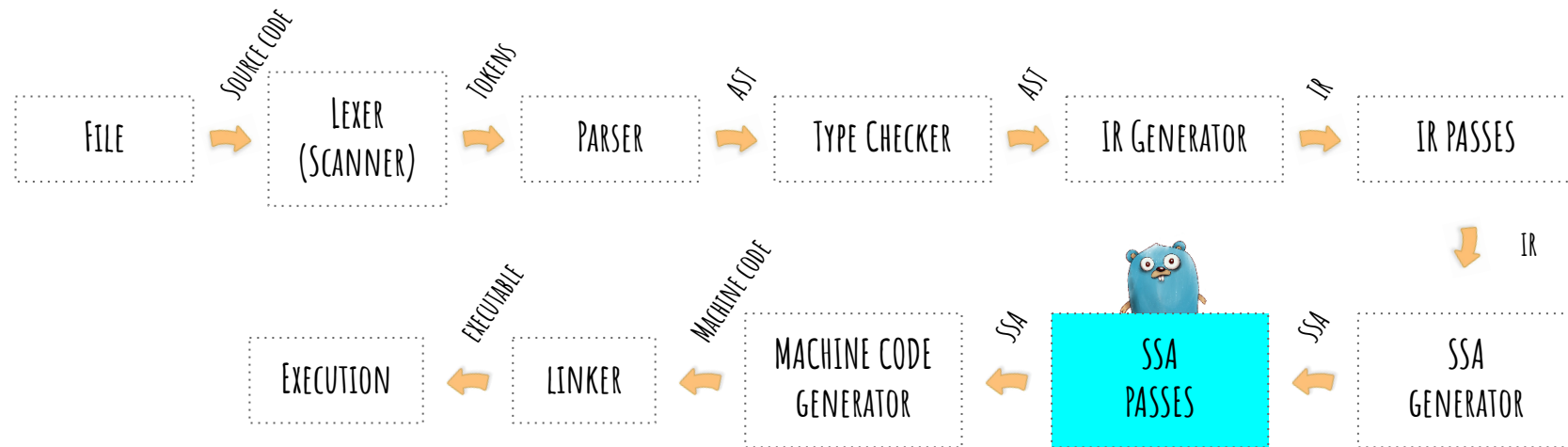
```
func (s *state) stmt(n ir.Node) {
```

```
case ir.ODCL:  
    n := n.(*ir.Decl)  
    if v := n.X; v.Esc() == ir.EscHeap {  
        s.newHeapaddr(v)  
    }
```

```
case ir.OVARDEF:  
    n := n.(*ir.UnaryExpr)  
    if !s.canSSA(n.X) {  
        s.vars[memVar] = s.newValue1Apos(ssa.OpVarDef, types.TypeMem, n.X.(*ir.Name), s.mem(), false)  
    }
```

```
case ir.OIF:  
    n := n.(*ir.IfStmt)  
    if ir.IsConst(n.Cond, constant.Bool) {  
        s.stmtList(n.Cond.Init())  
        if ir.BoolVal(n.Cond) {  
            s.stmtList(n.Body)  
        } else {  
            s.stmtList(n.Else)  
        }  
        break  
    }  
  
    bEnd := s.f.NewBlock(ssa.BlockPlain)  
    var likely int8  
    if n.Likely {  
        likely = 1  
    }  
    var bThen *ssa.Block  
    if len(n.Body) != 0 {  
        bThen = s.f.NewBlock(ssa.BlockPlain)  
    } else {  
        bThen = bEnd  
    }  
    var bElse *ssa.Block  
    if len(n.Else) != 0 {  
        bElse = s.f.NewBlock(ssa.BlockPlain)  
    } else {  
        bElse = bEnd  
    }  
    s.condBranch(n.Cond, bThen, bElse, likely)  
  
    if len(n.Body) != 0 {  
        s.startBlock(bThen)  
        s.stmtList(n.Body)  
        if b := s.endBlock(); b != nil {  
            b.AddEdgeTo(bEnd)  
        }  
    }  
    if len(n.Else) != 0 {  
        s.startBlock(bElse)  
        s.stmtList(n.Else)  
        if b := s.endBlock(); b != nil {  
            b.AddEdgeTo(bEnd)  
        }  
    }  
    s.startBlock(bEnd)
```

# SSA PASSES



# SSA PASSES

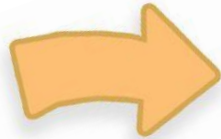
- deadcode
- shortcircuit
- cse
- lower
- and a lot more

# SSA (BEFORE THE PASSES)

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



```
b1:  
v1 (?) = InitMem <mem>  
v2 (?) = SP <uintptr>  
v3 (?) = SB <uintptr>  
v4 (?) = ConstInterface <any>  
v5 (?) = ArrayMake1 <[1]any> v4  
v6 (6) = VarDef <mem> {.autotmp_8} v1  
v7 (6) = LocalAddr <*[1]any> {.autotmp_8} v2 v6  
v8 (6) = Store <mem> {[1]any} v7 v5 v6  
v9 (6) = LocalAddr <*[1]any> {.autotmp_8} v2 v8  
v10 (?) = Addr <*uint8> {type.string} v3  
v11 (?) = Addr <*string> {main..stmp_0} v3  
v12 (6) = IMake <any> v10 v11  
v13 (6) = NilCheck <void> v9 v8  
v14 (?) = Const64 <int> [0] (fmt.n[int], fmt..autotmp_0[int])  
v15 (?) = Const64 <int> [1]  
v16 (6) = PtrIndex <*any> v9 v14  
v17 (6) = Store <mem> {any} v16 v12 v8  
v18 (6) = NilCheck <void> v9 v17  
v19 (6) = Copy <*any> v9  
v20 (6) = IsSliceInBounds <bool> v14 v15  
v25 (?) = ConstInterface <error> (fmt.err[error], fmt..autotmp_1[error])  
v28 (?) = Addr <*uint8> {go.itab.*os.File,io.Writer} v3  
v29 (?) = Addr <**os.File> {os.Stdout} v3  
If v20 → b2 b3 (likely) (6)  
  
b2: ← b1  
v23 (6) = Sub64 <int> v15 v14  
v24 (6) = SliceMake <[ ]any> v19 v23 v23 (fmt.a[[ ]any])  
v26 (6) = Copy <mem> v17  
v27 (+6) = InlMark <void> [0] v26  
v30 (294) = Load <*os.File> v29 v26  
v31 (294) = IMake <io.Writer> v28 v30  
v32 (294) = StaticLECall <int,error,mem> {AuxCall{fmt.Fprintln}} [40] v31 v24 v26  
v33 (294) = SelectN <mem> [2] v32  
v34 (294) = SelectN <int> [0] v32  
v35 (294) = SelectN <int> [0] v32 (fmt.n[int], fmt..autotmp_0[int])  
v36 (294) = SelectN <error> [1] v32 (fmt.err[error], fmt..autotmp_1[error])  
Plain → b4 (+6)  
  
b3: ← b1  
  
v21 (6) = Copy <mem> v17  
v22 (6) = PanicBounds <mem> [6] v14 v15 v21  
Exit v22 (6)  
  
b4: ← b2  
  
v38 (7) = Copy <mem> v33  
v37 (7) = MakeResult <mem> v38  
Ret v37 (7)  
  
name fmt.a[[ ]any]: v24  
name fmt.n[int]: v14 v35  
name fmt.err[error]: v25 v36  
name fmt..autotmp_0[int]: v14 v35  
name fmt..autotmp_1[error]: v25 v36
```

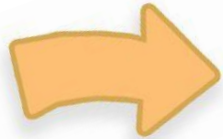
Generated with GOSSAFUNC=main.main go build hello.go

# SSA (AFTER THE PASSES)

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello world!")  
}
```



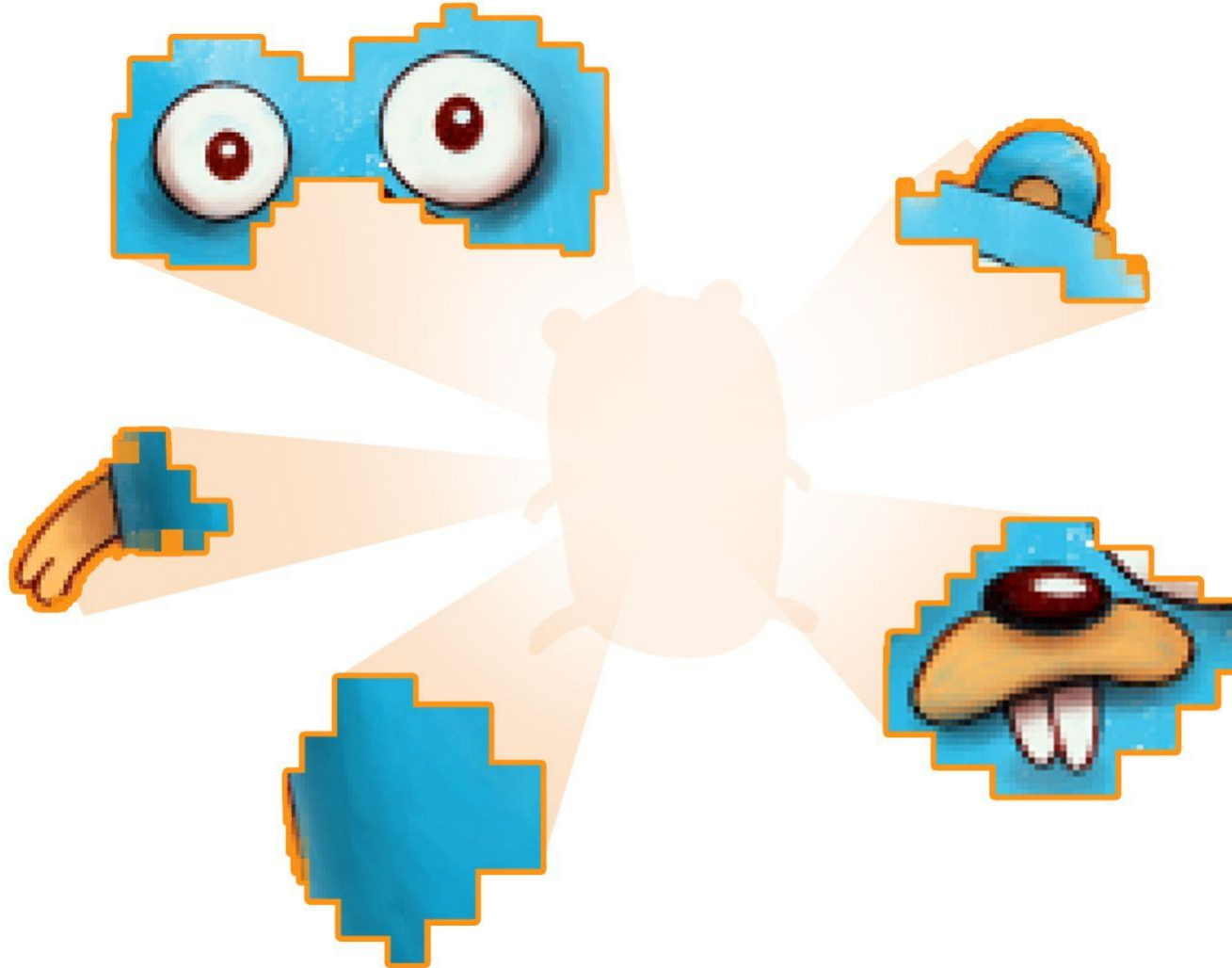
b4:

```
v1 (?) = InitMem <mem>  
v6 (6) = VarDef <mem> {.autotmp_8} v1  
v2 (?) = SP <uintptr> : SP  
v35 (6) = MOVQstoreconst <mem> {.autotmp_8} [val=0,off=0] v2 v6  
v3 (?) = SB <uintptr> : SB  
v14 (6) = LEAQ <*uint8> {type.string} v3 : DX  
v5 (+6) = MOVQstore <mem> {.autotmp_8} v2 v14 v35  
v20 (6) = LEAQ <*string> {main..stmp_0} v3 : DX  
v36 (+6) = MOVQstore <mem> {.autotmp_8} [8] v2 v20 v5  
v27 (+6) = InlMark <void> [0] v36  
v30 (+294) = MOVQload <*os.File> {os.Stdout} v3 v36 : BX  
v22 (294) = LEAQ <*uint8> {go.itab.*os.File,io.Writer} v3 : AX  
v13 (294) = LEAQ <*[1]any> {.autotmp_8} v2 : CX  
v16 (294) = MOVQconst <int> [1] : DI  
v7 (294) = Copy <int> v16 : SI  
v32 (294) = CALLstatic <int,unsafe.Pointer,unsafe.Pointer,mem> {AuxCall{fmt.Fprintln}} [40]...  
v33 (294) = SelectN <mem> [3] v32  
v37 (+7) = MakeResult <mem> v33
```

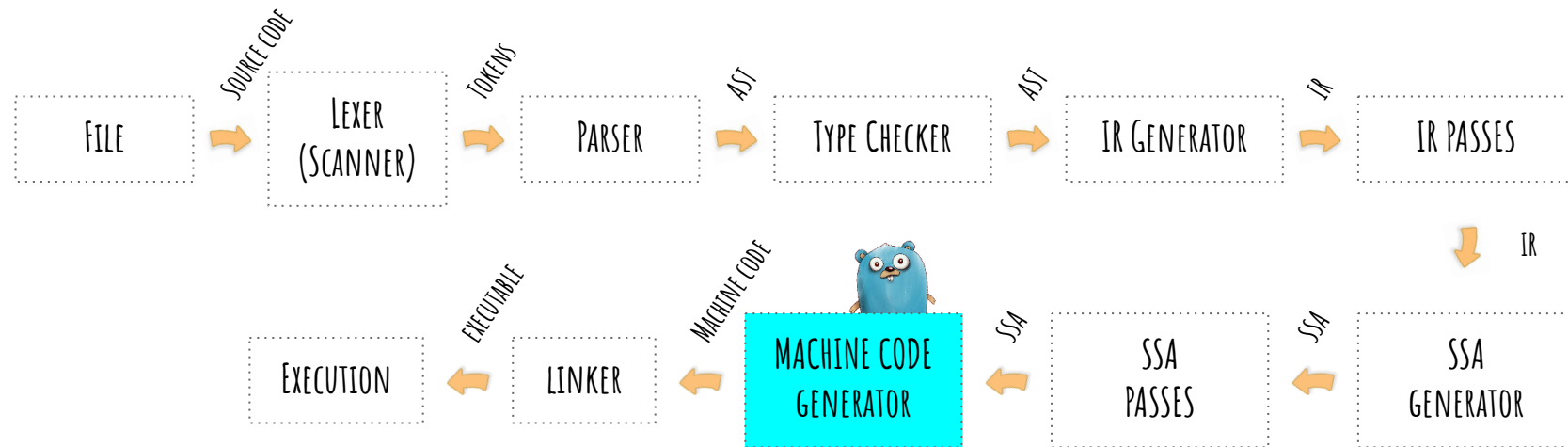
Ret v37 (7)

```
name a.ptr[*any]: v9  
name a.len[int]: v15  
name a.cap[int]: v15
```

# MACHINE CODE GENERATION



# MACHINE CODE GENERATION



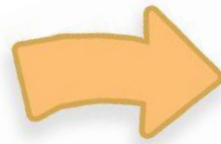


# MACHINE CODE GENERATION

b4:

```
v1 (?) = InitMem <mem>
v6 (6) = VarDef <mem> {.autotmp_8} v1
v2 (?) = SP <uintptr> : SP
v35 (6) = MOVOSToreconst <mem> {.autotmp_8} [val=0,off=0] v2 v6
v3 (?) = SB <uintptr> : SB
v14 (6) = LEAQ <*uint8> {type.string} v3 : DX
v5 (+6) = MOVQstore <mem> {.autotmp_8} v2 v14 v35
v20 (6) = LEAQ <*string> {main..stmp_0} v3 : DX
v36 (+6) = MOVQstore <mem> {.autotmp_8} [8] v2 v20 v5
v27 (+6) = InlMark <void> [0] v36
v30 (+294) = MOVQload <*os.File> {os.Stdout} v3 v36 : BX
v22 (294) = LEAQ <*uint8> {go.itab.*os.File,io.Writer} v3 : AX
v13 (294) = LEAQ <*[1]any> {.autotmp_8} v2 : CX
v16 (294) = MOVQconst <int> [1] : DI
v7 (294) = Copy <int> v16 : SI
v32 (294) = CALLstatic <int,unsafe.Pointer,unsafe.Pointer,mem>
{AuxCall{fmt.Fprintln}} [40]...
v33 (294) = SelectN <mem> [3] v32
v37 (+7) = MakeResult <mem> v33
Ret v37 (7)

name a.ptr[*any]: v9
name a.len[int]: v15
name a.cap[int]: v15
```



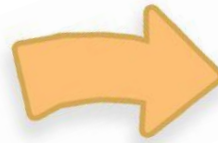
```
# /hello.go
00000 (5) TEXT main.main(SB), ABIInternal
00001 (5) FUNCDATA $0, gclocals·g2BeySu+wFnoycgXfElmcg==(SB)
00002 (5) FUNCDATA $1, gclocals·EaPwxsZ75yY1hHmVZLmk6g==(SB)
00003 (5) FUNCDATA $2, main.main.stkobj(SB)
00004 (+6) MOVUPS X15, main..autotmp_8-16(SP)
v35 00005 (6) LEAQ type.string(SB), DX
v14 00006 (6) MOVQ DX, main..autotmp_8-16(SP)
v5 00007 (6) LEAQ main..stmp_0(SB), DX
v20 00008 (6) MOVQ DX, main..autotmp_8-8(SP)
v36 00009 (?) NOP
v27
# $GOROOT/src/fmt/print.go
v30 00010 (+294) MOVQ os.Stdout(SB), BX
v22 00011 (294) LEAQ go.itab.*os.File,io.Writer(SB), AX
v13 00012 (294) LEAQ main..autotmp_8-16(SP), CX
v16 00013 (294) MOVL $1, DI
v7 00014 (294) MOVQ DI, SI
v32 00015 (294) PCDATA $1, $0
v32 00016 (294) CALL fmt.Fprintln(SB)
# /home/jespino/Projects/Github/go/hello.go
b4 00017 (7) RET
00018 (?) END
```

# MACHINE CODE GENERATION

b4:

```
v1 (?) = InitMem <mem>
v6 (6) = VarDef <mem> {.autotmp_8} v1
v2 (?) = SP <uintptr> : SP
v35 (6) = MOVOSToreconst <mem> {.autotmp_8} [val=0,off=0] v2 v6
v3 (?) = SB <uintptr> : SB
v14 (6) = LEAQ <*uint8> {type.string} v3 : DX
v5 (+6) = MOVQstore <mem> {.autotmp_8} v2 v14 v35
v20 (6) = LEAQ <*string> {main..stmp_0} v3 : DX
v36 (+6) = MOVQstore <mem> {.autotmp_8} [8] v2 v20 v5
v27 (+6) = InlMark <void> [0] v36
v30 (+294) = MOVQload <*os.File> {os.Stdout} v3 v36 : BX
v22 (294) = LEAQ <*uint8> {go.itab.*os.File,io.Writer} v3 : AX
v13 (294) = LEAQ <*[1]any> {.autotmp_8} v2 : CX
v16 (294) = MOVQconst <int> [1] : DI
v7 (294) = Copy <int> v16 : SI
v32 (294) = CALLstatic <int,unsafe.Pointer,unsafe.Pointer,mem>
{AuxCall{fmt.Fprintln}} [40]...
v33 (294) = SelectN <mem> [3] v32
v37 (+7) = MakeResult <mem> v33
Ret v37 (7)

name a.ptr[*any]: v9
name a.len[int]: v15
name a.cap[int]: v15
```



```
# /hello.go
00000 (5) TEXT main.main(SB), ABIInternal
00001 (5) FUNCDATA $0, gclocals·g2BeySu+wFnoycgXfElmcg== (SB)
00002 (5) FUNCDATA $1, gclocals·EaPwxsZ75yY1hHmVZLmk6g== (SB)
00003 (5) FUNCDATA $2, main.main.stkobj(SB)
00004 (+6) MOVUPS X15, main..autotmp_8-16(SP)
v35 00005 (6) LEAQ type.string(SB), DX
v14 00006 (6) MOVQ DX, main..autotmp_8-16(SP)
v5 00007 (6) LEAQ main..stmp_0(SB), DX
v20 00008 (6) MOVQ DX, main..autotmp_8-8(SP)
v36 00009 (?) NOP
v27
# $GOROOT/src/fmt/print.go
v30 00010 (+294) MOVQ os.Stdout(SB), BX
v22 00011 (294) LEAQ go.itab.*os.File,io.Writer(SB), AX
v13 00012 (294) LEAQ main..autotmp_8-16(SP), CX
v16 00013 (294) MOVL $1, DI
v7 00014 (294) MOVQ DI, SI
v32 00015 (294) PCDATA $1, $0
v32 00016 (294) CALL fmt.Fprintln(SB)
# /home/jespino/Projects/Github/go/hello.go
b4 00017 (7) RET
00018 (?) END
```

# MACHINE CODE GENERATION

```
case ssa.OpAMD64MOVQload, ssa.OpAMD64MOVLload, ssa.OpAMD64MOVWload, ssa.OpAMD64MOVBload, ssa.OpAMD64MOVOLoad,
ssa.OpAMD64MOVSSload, ssa.OpAMD64MOVSDload, ssa.OpAMD64MOVBSXload, ssa.OpAMD64MOVWQSXload, ssa.OpAMD64MOVLQSXload,
ssa.OpAMD64MOVBEQload, ssa.OpAMD64MOVBELload:
p := s.Prog(v.Op.Asm())
p.From.Type = obj.TYPE_MEM
p.From.Reg = v.Args[0].Reg()
ssagen.AddAux(&p.From, v)
p.To.Type = obj.TYPE_REG
p.To.Reg = v.Reg()
```

```
case ssa.OpAMD64CALLstatic, ssa.OpAMD64CALLtail:
if s.ABI == obj.ABI0 && v.Aux.(*ssa.AuxCall).Fn.ABI() == obj.ABIInternal {
    // zeroing X15 when entering ABIInternal from ABI0
    if buildcfg.GOOS != "plan9" { // do not use SSE on Plan 9
        opregreg(s, x86.AXORPS, x86.REG_X15, x86.REG_X15)
    }
    // set G register from TLS
    getgFromTLS(s, x86.REG_R14)
}
if v.Op == ssa.OpAMD64CALLtail {
    s.TailCall(v)
    break
}
s.Call(v)
if s.ABI == obj.ABIInternal && v.Aux.(*ssa.AuxCall).Fn.ABI() == obj.ABI0 {
    // zeroing X15 when entering ABIInternal from ABI0
    if buildcfg.GOOS != "plan9" { // do not use SSE on Plan 9
        opregreg(s, x86.AXORPS, x86.REG_X15, x86.REG_X15)
    }
    // set G register from TLS
    getgFromTLS(s, x86.REG_R14)
}
```

```
// Call returns a new CALL instruction for the SSA value v.
// It uses PrepareCall to prepare the call.
func (s *State) Call(v *ssa.Value) *obj.Prog {
    pPosIsStmt := s.pp.Pos.IsStmt() // The statement-ness fo the call comes from ssaGenState
    s.PrepareCall(v)

    p := s.Prog(obj.ACALL)
    if pPosIsStmt == src.PosIsStmt {
        p.Pos = v.Pos.WithIsStmt()
    } else {
        p.Pos = v.Pos.WithNotStmt()
    }
    if sym, ok := v.Aux.(*ssa.AuxCall); ok && sym.Fn != nil {
        p.To.Type = obj.TYPE_MEM
        p.To.Name = obj.NAME_EXTERN
        p.To.Sym = sym.Fn
    } else {
        // TODO(mdempsky): Can these differences be eliminated?
        switch Arch.LinkArch.Family {
        case sys.AMD64, sys.I386, sys.PPC64, sys.RISCV64, sys.S390X, sys.Wasm:
            p.To.Type = obj.TYPE_REG
        case sys.ARM, sys.ARM64, sys.Loong64, sys.MIPS, sys.MIPS64:
            p.To.Type = obj.TYPE_MEM
        default:
            base.Fatalf("unknown indirect call family")
        }
        p.To.Reg = v.Args[0].Reg()
    }
    return p
}
```

src/cmd/compile/internal/amd64/ssa.go:792  
src/cmd/compile/internal/amd64/ssa.go:1064  
src/cmd/compile/internal/ssagen/ssa.go:7603

# MACHINE CODE GENERATION

```
case ssa.OpAMD64MOVQload, ssa.OpAMD64MOVLload, ssa.OpAMD64MOVWload, ssa.OpAMD64MOVBload, ssa.OpAMD64MOV0load,
ssa.OpAMD64MOVSSload, ssa.OpAMD64MOVSDload, ssa.OpAMD64MOVBSXload, ssa.OpAMD64MOVWQSXload, ssa.OpAMD64MOVLQSXload,
ssa.OpAMD64MOVBEQload, ssa.OpAMD64MOVBELload:
```

```
p := s.Prog(v.Op.Asm())
p.From.Type = obj.TYPE_MEM
p.From.Reg = v.Args[0].Reg()
ssagen.AddAux(&p.From, v)
p.To.Type = obj.TYPE_REG
p.To.Reg = v.Reg()
```

```
case ssa.OpAMD64CALLstatic, ssa.OpAMD64CALLtail:
```

```
if s.ABI == obj.ABI0 && v.Aux.(*ssa.AuxCall).Fn.ABI() == obj.ABIInternal {
    // zeroing X15 when entering ABIInternal from ABI0
    if buildcfg.GOOS != "plan9" { // do not use SSE on Plan 9
        opregreg(s, x86.AXORPS, x86.REG_X15, x86.REG_X15)
    }
    // set G register from TLS
    getgFromTLS(s, x86.REG_R14)
}
if v.Op == ssa.OpAMD64CALLtail {
    s.TailCall(v)
    break
}
s.Call(v)
if s.ABI == obj.ABIInternal && v.Aux.(*ssa.AuxCall).Fn.ABI() == obj.ABI0 {
    // zeroing X15 when entering ABIInternal from ABI0
    if buildcfg.GOOS != "plan9" { // do not use SSE on Plan 9
        opregreg(s, x86.AXORPS, x86.REG_X15, x86.REG_X15)
    }
    // set G register from TLS
    getgFromTLS(s, x86.REG_R14)
}
```

```
// Call returns a new CALL instruction for the SSA value v.
// It uses PrepareCall to prepare the call.
```

```
func (s *State) Call(v *ssa.Value) *obj.Prog {
    pPosIsStmt := s.pp.Pos.IsStmt() // The statement-ness fo the call comes from ssaGenState
    s.PrepareCall(v)
```

```
p := s.Prog(obj.ACALL)
if pPosIsStmt == src.PosIsStmt {
    p.Pos = v.Pos.WithIsStmt()
} else {
    p.Pos = v.Pos.WithNotStmt()
}
if sym, ok := v.Aux.(*ssa.AuxCall); ok && sym.Fn != nil {
    p.To.Type = obj.TYPE_MEM
    p.To.Name = obj.NAME_EXTERN
    p.To.Sym = sym.Fn
} else {
    // TODO(mdempsky): Can these differences be eliminated?
    switch Arch.LinkArch.Family {
    case sys.AMD64, sys.I386, sys.PPC64, sys.RISCV64, sys.S390X, sys.Wasm:
        p.To.Type = obj.TYPE_REG
    case sys.ARM, sys.ARM64, sys.Loong64, sys.MIPS, sys.MIPS64:
        p.To.Type = obj.TYPE_MEM
    default:
        base.Fatalf("unknown indirect call family")
    }
    p.To.Reg = v.Args[0].Reg()
}
return p
}
```

```
src/cmd/compile/internal/amd64/ssa.go:792
src/cmd/compile/internal/amd64/ssa.go:1064
src/cmd/compile/internal/ssagen/ssa.go:7603
```

# MACHINE CODE GENERATION

```
case ssa.OpAMD64MOVQload, ssa.OpAMD64MOVLload, ssa.OpAMD64MOVWload, ssa.OpAMD64MOVBload, ssa.OpAMD64MOV0load,
ssa.OpAMD64MOVSSload, ssa.OpAMD64MOVSDload, ssa.OpAMD64MOVBSXload, ssa.OpAMD64MOVWQSXload, ssa.OpAMD64MOVLQSXload,
ssa.OpAMD64MOVBEQload, ssa.OpAMD64MOVBELload:
    p := s.Prog(v.Op.Asm())
    p.From.Type = obj.TYPE_MEM
    p.From.Reg = v.Args[0].Reg()
    ssagen.AddAux(&p.From, v)
    p.To.Type = obj.TYPE_REG
    p.To.Reg = v.Reg()
```

```
case ssa.OpAMD64CALLstatic, ssa.OpAMD64CALLtail:
    if s.ABI == obj.ABI0 && v.Aux.(*ssa.AuxCall).Fn.ABI() == obj.ABIInternal {
        // zeroing X15 when entering ABIInternal from ABI0
        if buildcfg.GOOS != "plan9" { // do not use SSE on Plan 9
            opregreg(s, x86.AXORPS, x86.REG_X15, x86.REG_X15)
        }
        // set G register from TLS
        getgFromTLS(s, x86.REG_R14)
    }
    if v.Op == ssa.OpAMD64CALLtail {
        s.TailCall(v)
        break
    }
    s.Call(v)
    if s.ABI == obj.ABIInternal && v.Aux.(*ssa.AuxCall).Fn.ABI() == obj.ABI0 {
        // zeroing X15 when entering ABIInternal from ABI0
        if buildcfg.GOOS != "plan9" { // do not use SSE on Plan 9
            opregreg(s, x86.AXORPS, x86.REG_X15, x86.REG_X15)
        }
        // set G register from TLS
        getgFromTLS(s, x86.REG_R14)
    }
}
```

```
// Call returns a new CALL instruction for the SSA value v.
// It uses PrepareCall to prepare the call.
func (s *State) Call(v *ssa.Value) *obj.Prog {
    pPosIsStmt := s.pp.Pos.IsStmt() // The statement-ness fo the call comes from ssaGenState
    s.PrepareCall(v)

    p := s.Prog(obj.ACALL)
    if pPosIsStmt == src.PosIsStmt {
        p.Pos = v.Pos.WithIsStmt()
    } else {
        p.Pos = v.Pos.WithNotStmt()
    }
    if sym, ok := v.Aux.(*ssa.AuxCall); ok && sym.Fn != nil {
        p.To.Type = obj.TYPE_MEM
        p.To.Name = obj.NAME_EXTERN
        p.To.Sym = sym.Fn
    } else {
        // TODO(mdempsky): Can these differences be eliminated?
        switch Arch.LinkArch.Family {
        case sys.AMD64, sys.I386, sys.PPC64, sys.RISCV64, sys.S390X, sys.Wasm:
            p.To.Type = obj.TYPE_REG
        case sys.ARM, sys.ARM64, sys.Loong64, sys.MIPS, sys.MIPS64:
            p.To.Type = obj.TYPE_MEM
        default:
            base.Fatalf("unknown indirect call family")
        }
        p.To.Reg = v.Args[0].Reg()
    }
    return p
}
```

src/cmd/compile/internal/amd64/ssa.go:792  
src/cmd/compile/internal/amd64/ssa.go:1064  
src/cmd/compile/internal/ssagen/ssa.go:7603

# MACHINE CODE GENERATION

```
case ssa.OpAMD64MOVQload, ssa.OpAMD64MOVLload, ssa.OpAMD64MOVWload, ssa.OpAMD64MOVBload, ssa.OpAMD64MOV0load,
ssa.OpAMD64MOVSSload, ssa.OpAMD64MOVSDload, ssa.OpAMD64MOVBSXload, ssa.OpAMD64MOVWQSXload, ssa.OpAMD64MOVLQSXload,
ssa.OpAMD64MOVBEQload, ssa.OpAMD64MOVBELload:
p := s.Prog(v.Op.Asm())
p.From.Type = obj.TYPE_MEM
p.From.Reg = v.Args[0].Reg()
ssagen.AddAux(&p.From, v)
p.To.Type = obj.TYPE_REG
p.To.Reg = v.Reg()
```

```
case ssa.OpAMD64CALLstatic, ssa.OpAMD64CALLtail:
if s.ABI == obj.ABI0 && v.Aux.(*ssa.AuxCall).Fn.ABI() == obj.ABIInternal {
    // zeroing X15 when entering ABIInternal from ABI0
    if buildcfg.GOOS != "plan9" { // do not use SSE on Plan 9
        opregreg(s, x86.AXORPS, x86.REG_X15, x86.REG_X15)
    }
    // set G register from TLS
    getgFromTLS(s, x86.REG_R14)
}
if v.Op == ssa.OpAMD64CALLtail {
    s.TailCall(v)
    break
}
s.Call(v)
if s.ABI == obj.ABIInternal && v.Aux.(*ssa.AuxCall).Fn.ABI() == obj.ABI0 {
    // zeroing X15 when entering ABIInternal from ABI0
    if buildcfg.GOOS != "plan9" { // do not use SSE on Plan 9
        opregreg(s, x86.AXORPS, x86.REG_X15, x86.REG_X15)
    }
    // set G register from TLS
    getgFromTLS(s, x86.REG_R14)
}
```

```
// Call returns a new CALL instruction for the SSA value v.
// It uses PrepareCall to prepare the call.
func (s *State) Call(v *ssa.Value) *obj.Prog {
    pPosIsStmt := s.pp.Pos.IsStmt() // The statement-ness fo the call comes from ssaGenState
    s.PrepareCall(v)

    p := s.Prog(obj.ACALL)
    if pPosIsStmt == src.PosIsStmt {
        p.Pos = v.Pos.WithIsStmt()
    } else {
        p.Pos = v.Pos.WithNotStmt()
    }
    if sym, ok := v.Aux.(*ssa.AuxCall); ok && sym.Fn != nil {
        p.To.Type = obj.TYPE_MEM
        p.To.Name = obj.NAME_EXTERN
        p.To.Sym = sym.Fn
    } else {
        // TODO(mdempsky): Can these differences be eliminated?
        switch Arch.LinkArch.Family {
        case sys.AMD64, sys.I386, sys.PPC64, sys.RISCV64, sys.S390X, sys.Wasm:
            p.To.Type = obj.TYPE_REG
        case sys.ARM, sys.ARM64, sys.Loong64, sys.MIPS, sys.MIPS64:
            p.To.Type = obj.TYPE_MEM
        default:
            base.Fatalf("unknown indirect call family")
        }
        p.To.Reg = v.Args[0].Reg()
    }
    return p
}
```

src/cmd/compile/internal/amd64/ssa.go:792  
src/cmd/compile/internal/amd64/ssa.go:1064  
src/cmd/compile/internal/ssagen/ssa.go:7603

# MACHINE CODE GENERATION

```
case ssa.OpAMD64MOVQload, ssa.OpAMD64MOVLload, ssa.OpAMD64MOVWload, ssa.OpAMD64MOVBload, ssa.OpAMD64MOVOLoad,
ssa.OpAMD64MOVSSload, ssa.OpAMD64MOVSDload, ssa.OpAMD64MOVBSXload, ssa.OpAMD64MOVWQSXload, ssa.OpAMD64MOVLQSXload,
ssa.OpAMD64MOVBEQload, ssa.OpAMD64MOVBELload:
p := s.Prog(v.Op.Asm())
p.From.Type = obj.TYPE_MEM
p.From.Reg = v.Args[0].Reg()
ssagen.AddAux(&p.From, v)
p.To.Type = obj.TYPE_REG
p.To.Reg = v.Reg()
```

```
case ssa.OpAMD64CALLstatic, ssa.OpAMD64CALLtail:
if s.ABI == obj.ABI0 && v.Aux.(*ssa.AuxCall).Fn.ABI() == obj.ABIInternal {
    // zeroing X15 when entering ABIInternal from ABI0
    if buildcfg.GOOS != "plan9" { // do not use SSE on Plan 9
        opregreg(s, x86.AXORPS, x86.REG_X15, x86.REG_X15)
    }
    // set G register from TLS
    getgFromTLS(s, x86.REG_R14)
}
if v.Op == ssa.OpAMD64CALLtail {
    s.TailCall(v)
    break
}
s.Call(v)
if s.ABI == obj.ABIInternal && v.Aux.(*ssa.AuxCall).Fn.ABI() == obj.ABI0 {
    // zeroing X15 when entering ABIInternal from ABI0
    if buildcfg.GOOS != "plan9" { // do not use SSE on Plan 9
        opregreg(s, x86.AXORPS, x86.REG_X15, x86.REG_X15)
    }
    // set G register from TLS
    getgFromTLS(s, x86.REG_R14)
}
```

```
// Call returns a new CALL instruction for the SSA value v.
// It uses PrepareCall to prepare the call.
func (s *State) Call(v *ssa.Value) *obj.Prog {
    pPosIsStmt := s.pp.Pos.IsStmt() // The statement-ness fo the call comes from ssaGenState
    s.PrepareCall(v)

    p := s.Prog(obj.ACALL)
    if pPosIsStmt == src.PosIsStmt {
        p.Pos = v.Pos.WithIsStmt()
    } else {
        p.Pos = v.Pos.WithNotStmt()
    }
    if sym, ok := v.Aux.(*ssa.AuxCall); ok && sym.Fn != nil {
        p.To.Type = obj.TYPE_MEM
        p.To.Name = obj.NAME_EXTERN
        p.To.Sym = sym.Fn
    } else {
        // TODO(mdempsky): Can these differences be eliminated?
        switch Arch.LinkArch.Family {
        case sys.AMD64, sys.I386, sys.PPC64, sys.RISCV64, sys.S390X, sys.Wasm:
            p.To.Type = obj.TYPE_REG
        case sys.ARM, sys.ARM64, sys.Loong64, sys.MIPS, sys.MIPS64:
            p.To.Type = obj.TYPE_MEM
        default:
            base.Fatalf("unknown indirect call family")
        }
        p.To.Reg = v.Args[0].Reg()
    }
    return p
}
```

src/cmd/compile/internal/amd64/ssa.go:792  
src/cmd/compile/internal/amd64/ssa.go:1064  
src/cmd/compile/internal/ssagen/ssa.go:7603

# MACHINE CODE GENERATION

```
case ssa.OpAMD64MOVQload, ssa.OpAMD64MOVLload, ssa.OpAMD64MOVWload, ssa.OpAMD64MOVBload, ssa.OpAMD64MOVOLoad,
ssa.OpAMD64MOVSSload, ssa.OpAMD64MOVSDload, ssa.OpAMD64MOVBSXload, ssa.OpAMD64MOVWQSXload, ssa.OpAMD64MOVLQSXload,
ssa.OpAMD64MOVBEQload, ssa.OpAMD64MOVBELload:
p := s.Prog(v.Op.Asm())
p.From.Type = obj.TYPE_MEM
p.From.Reg = v.Args[0].Reg()
ssagen.AddAux(&p.From, v)
p.To.Type = obj.TYPE_REG
p.To.Reg = v.Reg()
```

```
case ssa.OpAMD64CALLstatic, ssa.OpAMD64CALLtail:
if s.ABI == obj.ABI0 && v.Aux.(*ssa.AuxCall).Fn.ABI() == obj.ABIInternal {
    // zeroing X15 when entering ABIInternal from ABI0
    if buildcfg.GOOS != "plan9" { // do not use SSE on Plan 9
        opregreg(s, x86.AXORPS, x86.REG_X15, x86.REG_X15)
    }
    // set G register from TLS
    getgFromTLS(s, x86.REG_R14)
}
if v.Op == ssa.OpAMD64CALLtail {
    s.TailCall(v)
    break
}
s.Call(v)
if s.ABI == obj.ABIInternal && v.Aux.(*ssa.AuxCall).Fn.ABI() == obj.ABI0 {
    // zeroing X15 when entering ABIInternal from ABI0
    if buildcfg.GOOS != "plan9" { // do not use SSE on Plan 9
        opregreg(s, x86.AXORPS, x86.REG_X15, x86.REG_X15)
    }
    // set G register from TLS
    getgFromTLS(s, x86.REG_R14)
}
```

```
// Call returns a new CALL instruction for the SSA value v.
// It uses PrepareCall to prepare the call.
func (s *State) Call(v *ssa.Value) *obj.Prog {
    pPosIsStmt := s.pp.Pos.IsStmt() // The statement-ness fo the call comes from ssaGenState
    s.PrepareCall(v)

    p := s.Prog(obj.ACALL)
    if pPosIsStmt == src.PosIsStmt {
        p.Pos = v.Pos.WithIsStmt()
    } else {
        p.Pos = v.Pos.WithNotStmt()
    }
    if sym, ok := v.Aux.(*ssa.AuxCall); ok && sym.Fn != nil {
        p.To.Type = obj.TYPE_MEM
        p.To.Name = obj.NAME_EXTERN
        p.To.Sym = sym.Fn
    } else {
        // TODO(mdempsky): Can these differences be eliminated?
        switch Arch.LinkArch.Family {
        case sys.AMD64, sys.I386, sys.PPC64, sys.RISCV64, sys.S390X, sys.Wasm:
            p.To.Type = obj.TYPE_REG
        case sys.ARM, sys.ARM64, sys.Loong64, sys.MIPS, sys.MIPS64:
            p.To.Type = obj.TYPE_MEM
        default:
            base.Fatalf("unknown indirect call family")
        }
        p.To.Reg = v.Args[0].Reg()
    }
    return p
}
```

src/cmd/compile/internal/amd64/ssa.go:792  
src/cmd/compile/internal/amd64/ssa.go:1064  
src/cmd/compile/internal/ssagen/ssa.go:7603



# MACHINE CODE GENERATION

```
case ssa.OpAMD64MOVQload, ssa.OpAMD64MOVLload, ssa.OpAMD64MOVWload, ssa.OpAMD64MOVBload, ssa.OpAMD64MOVOLoad,
ssa.OpAMD64MOVSSload, ssa.OpAMD64MOVSDload, ssa.OpAMD64MOVBSXload, ssa.OpAMD64MOVWQSXload, ssa.OpAMD64MOVLQSXload,
ssa.OpAMD64MOVBEQload, ssa.OpAMD64MOVBELload:
p := s.Prog(v.Op.Asm())
p.From.Type = obj.TYPE_MEM
p.From.Reg = v.Args[0].Reg()
ssagen.AddAux(&p.From, v)
p.To.Type = obj.TYPE_REG
p.To.Reg = v.Reg()
```

```
case ssa.OpAMD64CALLstatic, ssa.OpAMD64CALLtail:
if s.ABI == obj.ABI0 && v.Aux.(*ssa.AuxCall).Fn.ABI() == obj.ABIInternal {
    // zeroing X15 when entering ABIInternal from ABI0
    if buildcfg.GOOS != "plan9" { // do not use SSE on Plan 9
        opregreg(s, x86.AXORPS, x86.REG_X15, x86.REG_X15)
    }
    // set G register from TLS
    getgFromTLS(s, x86.REG_R14)
}
if v.Op == ssa.OpAMD64CALLtail {
    s.TailCall(v)
    break
}
s.Call(v)
if s.ABI == obj.ABIInternal && v.Aux.(*ssa.AuxCall).Fn.ABI() == obj.ABI0 {
    // zeroing X15 when entering ABIInternal from ABI0
    if buildcfg.GOOS != "plan9" { // do not use SSE on Plan 9
        opregreg(s, x86.AXORPS, x86.REG_X15, x86.REG_X15)
    }
    // set G register from TLS
    getgFromTLS(s, x86.REG_R14)
}
```

```
// Call returns a new CALL instruction for the SSA value v.
// It uses PrepareCall to prepare the call.
func (s *State) Call(v *ssa.Value) *obj.Prog {
    pPosIsStmt := s.pp.Pos.IsStmt() // The statement-ness fo the call comes from ssaGenState
    s.PrepareCall(v)

    p := s.Prog(obj.ACALL)
    if pPosIsStmt == src.PosIsStmt {
        p.Pos = v.Pos.WithIsStmt()
    } else {
        p.Pos = v.Pos.WithNotStmt()
    }
    if sym, ok := v.Aux.(*ssa.AuxCall); ok && sym.Fn != nil {
        p.To.Type = obj.TYPE_MEM
        p.To.Name = obj.NAME_EXTERN
        p.To.Sym = sym.Fn
    } else {
        // TODO(mdempsky): Can these differences be eliminated?
        switch Arch.LinkArch.Family {
        case sys.AMD64, sys.I386, sys.PPC64, sys.RISCV64, sys.S390X, sys.Wasm:
            p.To.Type = obj.TYPE_REG
        case sys.ARM, sys.ARM64, sys.Loong64, sys.MIPS, sys.MIPS64:
            p.To.Type = obj.TYPE_MEM
        default:
            base.Fatalf("unknown indirect call family")
        }
        p.To.Reg = v.Args[0].Reg()
    }
    return p
}
```

src/cmd/compile/internal/amd64/ssa.go:792  
src/cmd/compile/internal/amd64/ssa.go:1064  
src/cmd/compile/internal/ssagen/ssa.go:7603

# MACHINE CODE GENERATION

```
case ssa.OpAMD64MOVQload, ssa.OpAMD64MOVLload, ssa.OpAMD64MOVWload, ssa.OpAMD64MOVBload, ssa.OpAMD64MOV0load,
ssa.OpAMD64MOVSSload, ssa.OpAMD64MOVSDload, ssa.OpAMD64MOVBSXload, ssa.OpAMD64MOVWQSXload, ssa.OpAMD64MOVLQSXload,
ssa.OpAMD64MOVBEQload, ssa.OpAMD64MOVBELload:
p := s.Prog(v.Op.Asm())
p.From.Type = obj.TYPE_MEM
p.From.Reg = v.Args[0].Reg()
ssagen.AddAux(&p.From, v)
p.To.Type = obj.TYPE_REG
p.To.Reg = v.Reg()
```

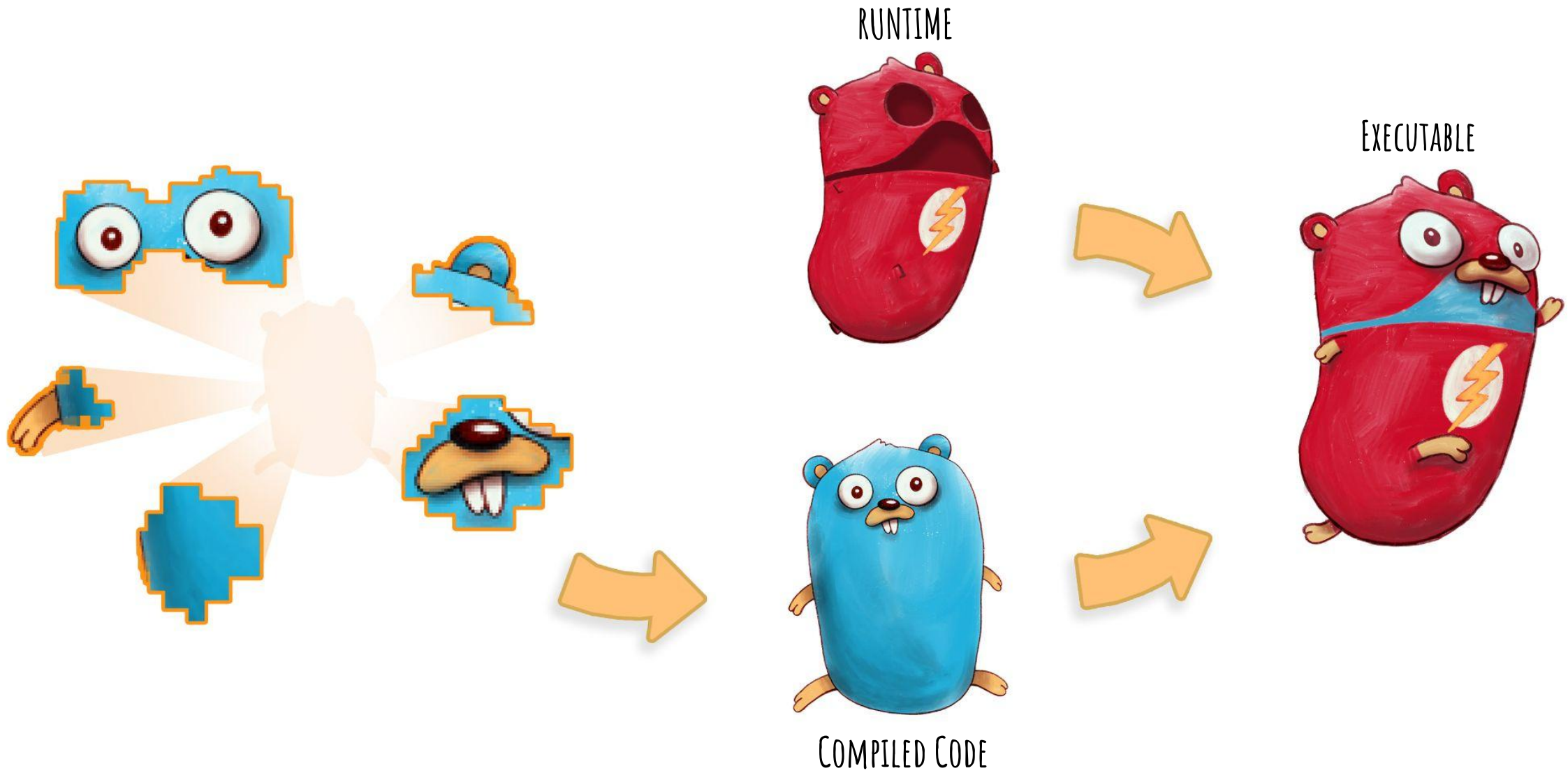
```
case ssa.OpAMD64CALLstatic, ssa.OpAMD64CALLtail:
if s.ABI == obj.ABI0 && v.Aux.(*ssa.AuxCall).Fn.ABI() == obj.ABIInternal {
    // zeroing X15 when entering ABIInternal from ABI0
    if buildcfg.GOOS != "plan9" { // do not use SSE on Plan 9
        opregreg(s, x86.AXORPS, x86.REG_X15, x86.REG_X15)
    }
    // set G register from TLS
    getgFromTLS(s, x86.REG_R14)
}
if v.Op == ssa.OpAMD64CALLtail {
    s.TailCall(v)
    break
}
s.Call(v)
if s.ABI == obj.ABIInternal && v.Aux.(*ssa.AuxCall).Fn.ABI() == obj.ABI0 {
    // zeroing X15 when entering ABIInternal from ABI0
    if buildcfg.GOOS != "plan9" { // do not use SSE on Plan 9
        opregreg(s, x86.AXORPS, x86.REG_X15, x86.REG_X15)
    }
    // set G register from TLS
    getgFromTLS(s, x86.REG_R14)
}
```

```
// Call returns a new CALL instruction for the SSA value v.
// It uses PrepareCall to prepare the call.
func (s *State) Call(v *ssa.Value) *obj.Prog {
    pPosIsStmt := s.pp.Pos.IsStmt() // The statement-ness fo the call comes from ssaGenState
    s.PrepareCall(v)

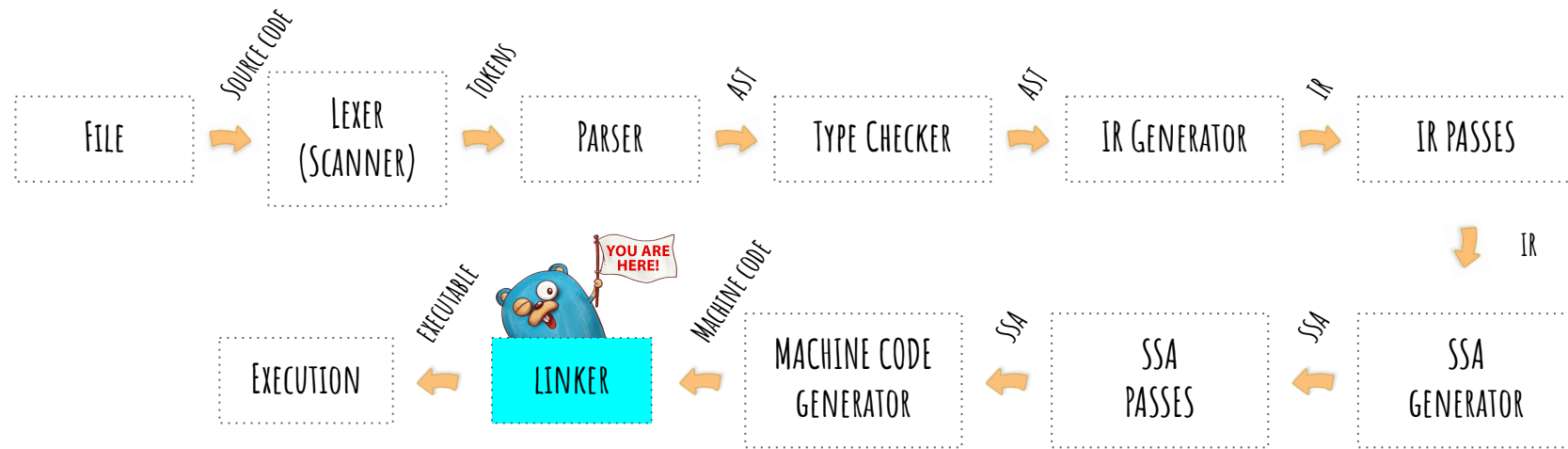
    p := s.Prog(obj.ACALL)
    if pPosIsStmt == src.PosIsStmt {
        p.Pos = v.Pos.WithIsStmt()
    } else {
        p.Pos = v.Pos.WithNotStmt()
    }
    if sym, ok := v.Aux.(*ssa.AuxCall); ok && sym.Fn != nil {
        p.To.Type = obj.TYPE_MEM
        p.To.Name = obj.NAME_EXTERN
        p.To.Sym = sym.Fn
    } else {
        // TODO(mdempsky): Can these differences be eliminated?
        switch Arch.LinkArch.Family {
        case sys.AMD64, sys.I386, sys.PPC64, sys.RISCV64, sys.S390X, sys.Wasm:
            p.To.Type = obj.TYPE_REG
        case sys.ARM, sys.ARM64, sys.Loong64, sys.MIPS, sys.MIPS64:
            p.To.Type = obj.TYPE_MEM
        default:
            base.Fatalf("unknown indirect call family")
        }
        p.To.Reg = v.Args[0].Reg()
    }
    return p
}
```

src/cmd/compile/internal/amd64/ssa.go:792  
src/cmd/compile/internal/amd64/ssa.go:1064  
src/cmd/compile/internal/ssagen/ssa.go:7603

# LINKING



# LINKING



# THE RUNTIME



- Maps, slices, channels, goroutines...
- Memory management
- The scheduler
- The startup



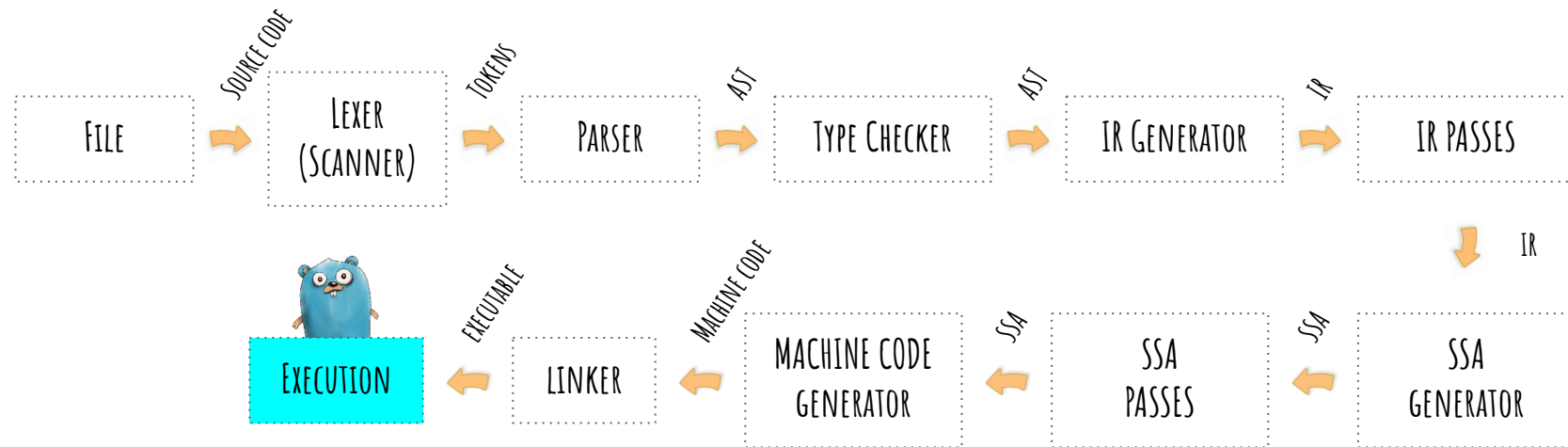


AND TO THE SCREEN

# THE KERNEL AND THE STDOUT



# THE KERNEL AND THE STDOUT





# PRINTING TO THE SCREEN

```
$ strace ./a.out
```

```
execve("./a.out", [ "./a.out" ], 0x7ffde58585d0 /* 77 vars */) = 0
arch_prctl(ARCH_SET_FS, 0x528870) = 0
sched_getaffinity(0, 8192, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]) = 40
openat(AT_FDCWD, "/sys/kernel/mm/transparent_hugepage/hpage_pmd_size", O_RDONLY) = 3
read(3, "2097152\n", 20) = 8
close(3) = 0
```

```
...
```

```
write(1, "hello-world!\n", 13hello-world!) = 13
exit_group(0) = ?
+++ exited with 0 +++
```

# PRINTING TO THE SCREEN

```
$ strace ./a.out
```

```
execve("./a.out", [ "./a.out" ], 0x7ffde58585d0 /* 77 vars */) = 0
arch_prctl(ARCH_SET_FS, 0x528870) = 0
sched_getaffinity(0, 8192, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]) = 40
openat(AT_FDCWD, "/sys/kernel/mm/transparent_hugepage/hpage_pmd_size", O_RDONLY) = 3
read(3, "2097152\n", 20) = 8
close(3) = 0
```

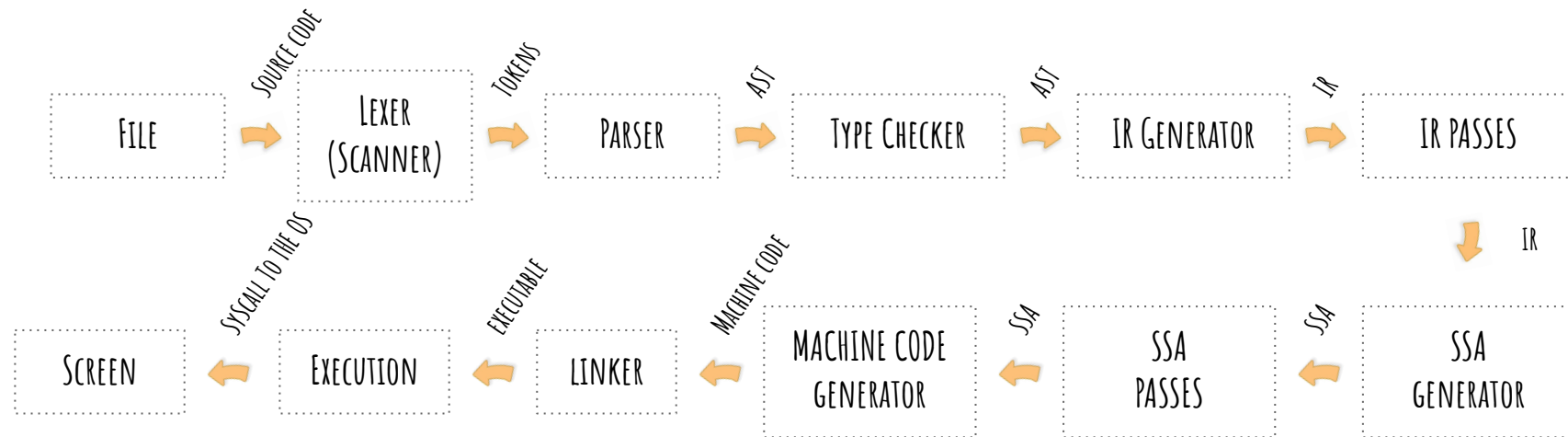
```
...
```

```
write(1, "hello-world!\n", 13hello-world!) = 13
exit_group(0) = ?
+++ exited with 0 +++
```



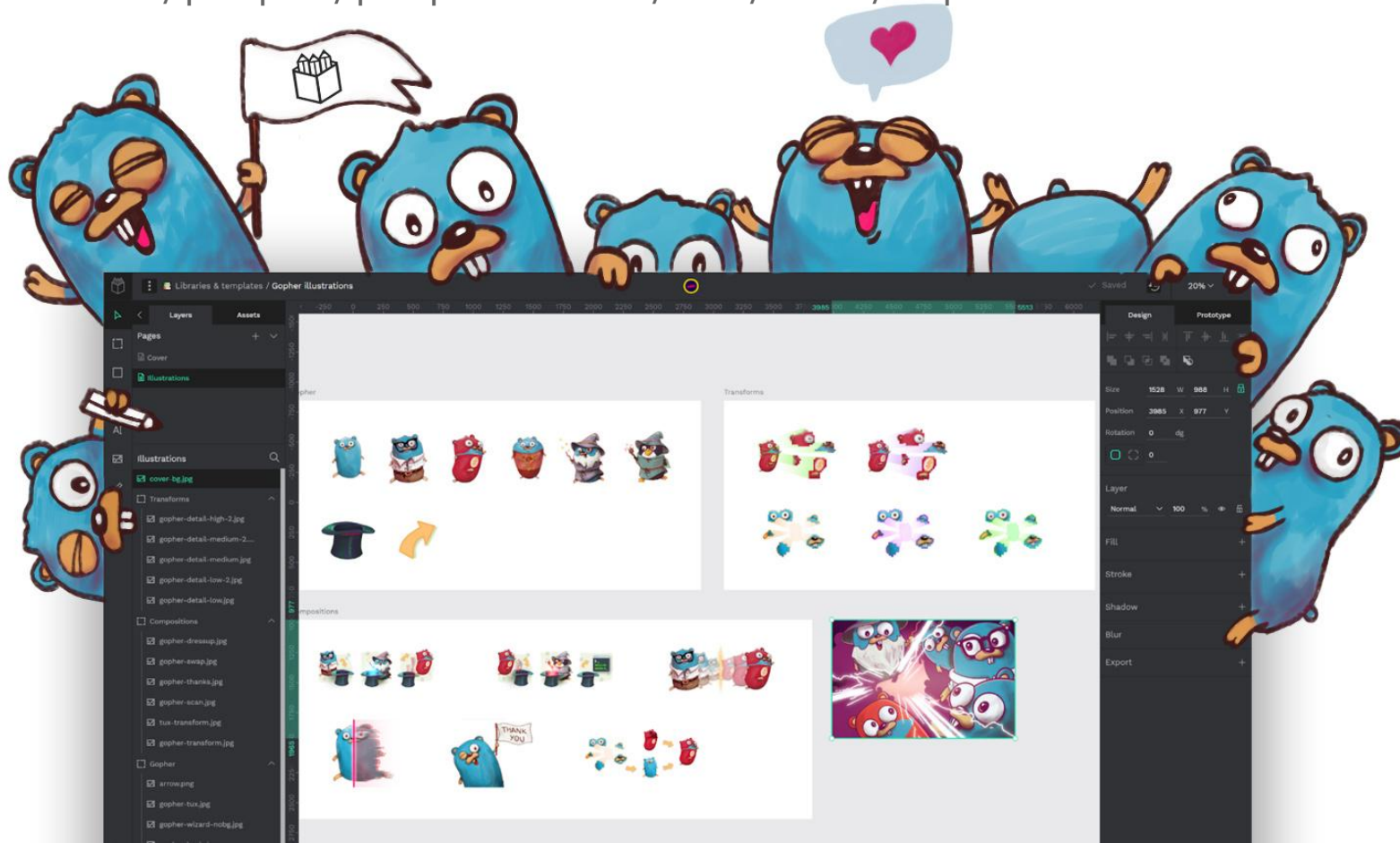
# SUMMARY

# SUMMARY



# THE ILLUSTRATIONS OF THE TALK

- Made by Juan de la Cruz for this talk
- Creative Commons 0 (Use it however you want)
- Downloadable in Penpot (Open Source Design tool) format
- <https://github.com/penpot/penpot-files/raw/main/Gopher-illustrations.penpot>





A GIFT FROM MATTERMOST



WWW.EUENYO



# REFERENCES

- The compile command README: <https://go.dev/src/cmd/compile/README>
- The SSA README: <https://go.dev/src/cmd/compile/internal/ssa/README>
- <https://go.dev/doc/asm>
- <https://pkg.go.dev/runtime>
- SSA Talks:
  - <https://www.youtube.com/watch?v=D2-gaMvWfQY>
  - <https://www.youtube.com/watch?v=uTMvKVma5ms>
- Go Assembler: <https://www.youtube.com/watch?v=KINIAgRpkDA>



# CONCLUSIONS



