COMP520 - Blank Identifier Specification

Alexander Krolik

February 9, 2020

1 Introduction

The blank identifier (i.e `_`) in `Go` is used to indicate “void” or “nothing”. Conceptually it is similar to `/dev/null`. This may be surprising, but it’s useful when combined with multiple return values.

```
func foo() int, int {
    return 0, 1
}
var a, _ = foo()
```

In `GoLite`, we don’t support multiple return values, but we maintain the blank identifier for language completeness. To ensure `GoLite` is a strict subset, this means verifying it is used correctly. Surprisingly, implementing this seemingly trivial functionality is fairly tricky and is the source of many bugs in the project. This document outlines the valid/invalid cases, as well as their implications.

2 Valid

Blank identifiers are valid as the identifier of declarations, and in the destination of assignments.

```
// Variable declaration
var _ int

// Type declarations
type _ int
type _ struct {
    _ int
}

// Function/parameter declarations
func _(_ int) {
}

// Short declaration
_, a := 0, 0

// Assignment
_ = 0
```

2.1 Bindings

Blank identifiers do not introduce bindings - meaning they are not entered into the symbol table and cannot be used as a value. The following example program is therefore valid, despite the same identifier (i.e. the blank identifier) being declared multiple times in the same scope.

```
package main

func _(_ int, _ float64) int {
    return 0
}

func _() float64 {
    return 0.0
}

func main() {
    var _ int
    var _ float64

    type _ int
    type _ struct {
        _ int
        _ float64
    }

    _, a := 0, 0
}
```

3 Invalid

As blank identifiers are never bound and cannot be referenced - they are neither value, type, nor function. This means there are numerous contexts under which they are invalid. Exhaustively:

- Package identifier
- Expressions which evaluate to a value
  - Binary expression operands
  - Unary expression operands
  - Array indexing target and index
  - Struct target and field
  - Function call identifier and parameters (including builtins)
  - Cast type and expression
- Lastly, the target of an assignment op (e.g. `+=`) or increment/decrement may not be a blank identifier (as it must read the value to change)

Interestingly, while blank function declarations and struct fields are valid, they may not be accessed.

```
func _() {
}

var a struct {
    _ int
}

func main() {
    _()     // Error: Cannot call blank function
    a._     // Error: Cannot access blank field
}
```

Notice that the correct use of the blank identifier can be expressed as either through syntax or type rules! The decision is left up to you - you may find it easier to encode the checks in the grammar, a weeder, or in the typechecker/symbol table. The latter is the approach in the official `Go` compiler, triggering an error whenever the type/value of a blank identifier is requested.

4 Typechecking

Regardless of your implementation choice, you must take into account the valid uses of the blank identifier in the symbol table and typechecker for your project. Importantly,

1. Blank identifiers are never entered into the symbol table
2. Blank identifiers may represent any type/kind of data (and may differ throughout the program)

5 Codegen

Generating code for blank identifiers in statically-typed languages is challenging. As languages like `C` and `C++` do not have the concept of a blank identifier, the following code will produce a compiler error. The same logic holds for function names, struct fields, parameters, and short declarations.

```
int _ = 0;
int _ = 1; // Error: _ already declared
```

We can however use a couple tricks to achieve the correct behaviour. Recall that the blank identifier cannot be accessed, meaning the storage of data into `_` is not actually required. For our purposes (`Go` may differ), this means we can eliminate blank function declarations and struct fields. We cannot, however, completely eliminate assignments. Consider the following code which calls a function `id`.

```
func id(a int) int {
    println(a)
    return a
}

func foo() {
    _ = id(1)
}
```

Since the RHS of the assignment is a function call with side-effects, we are required to execute the function and thus cannot remove the entire assignment. However, as the return value is never used, we can produce the following equivalent `C` code which evaluates the function and simply ignores the result.

```
int id(int a) {
    printf("%d\n", a);
    return a;
}

void foo() {
    id(1);
}
```

The reference compiler implements this behaviour using temporary variables which are never referenced. Each blank identifier is assigned a fresh temporary variable of the correct type, the assignment performed, and the variable never accessed again.

References

https://golang.org/ref/spec#Blank_identifier