Assignment 4 - Semantic Analysis

Ethan Benabou

ID: 40032543

1 - Semantic Rules

Below is the full list of semantic rules that have been implemented by the additions in this phase of the assignment.

Symbol Table Creation Phase

1. ✓ Global scope symbol table.
2. ✓ Individual symbol tables for each struct, impl, and top-level function. These tables are "nested" (linked to) from within their parent tables. E.g. the symbol table for struct `MyStruct` is linked to from the `MyStruct` entry in the global symbol table.
3. ✓ Symbol table entries for each scope data member (e.g. struct data members, function-local variables).
4. ✓ Function's define scopes with their own symbol table, these tables are linked to from the function's entry in a parent table.
5. ✓ The symbol table creation phase emits errors + warnings for certain semantic errors. Including: duplicate identifiers, warnings for shadowed members.
6. ✓ This phase also emits errors for declared but not defined functions, and defined but not declared functions.
7. ✓ Symbol table printouts are supported.
8. ✓ Errors are emitted for duplicate identifiers within the same scope.
9. ✓ Warnings are emitted for function overloading.

Semantic Checking Phase (Binding and Type Checking)

10. ✓ Expression, assignment, and return statement type checking is performed.
11. ✓ Errors emitted for usage of scope-undefined identifiers.
12. ✓ Function calls are type checked + number of parameters is checked, type checking of expressions used as function parameters is performed.
13. ✓ Array dimensions lists are checked.
14. ✓ Errors emitted for circular class dependencies.
15. ✓ `.` operator checked for correct usage. This operator should only be used on variables of a struct type, and only to access valid members of this struct (fields or struct methods). If this described usage is not obeyed, errors are emitted from the is checking phase.

2 - Design

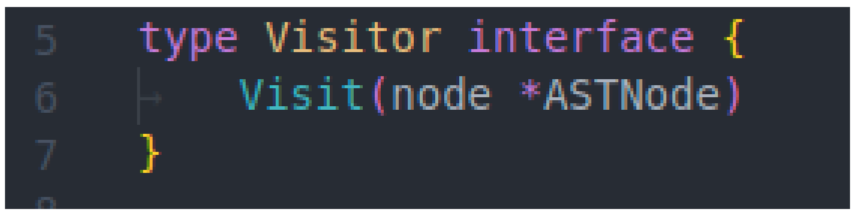
In this assignment, two post-parse semantic analysis phases are added to the compiler. These phases are, as described by prof Pacquet, the ***Symbol Table Creation Phase*** which traverses the AST datastructure emitted by the parser, depth-first, and assembles symbol tables for every scope within the program. This phase emits an annotated AST *intermediate representation* which we will call `SymAST`.

Also implemented is the ***Semantic Checking Phase***, which consumes the `SymAST` IR created by the symbol table creation phase. This phase executes further semantic checks to ensure that basic language semantics are obeyed. In particular, binding and type checking semantic actions are performed as the `SymAST` is traversed. This phase emits an annotated SymAST *intermediate representation* which we will call `TypeSymAST`

With both of these phases implemented, the compiler processing pipeline is roughly:

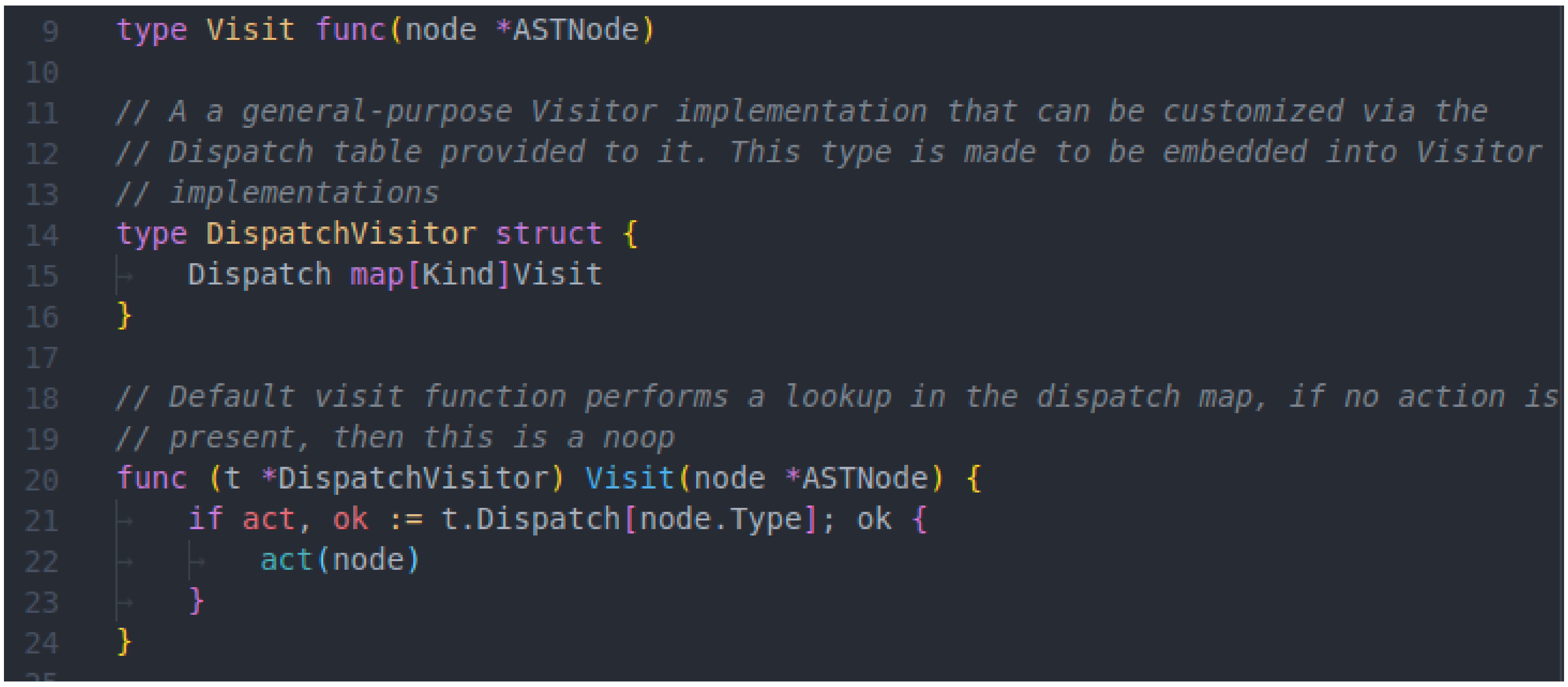
```
source    -->    AST    -->    SymAST    -->    TypeSymAST
        PARSER       SYMBOL          TYPE
                     TABLE           CHECK
                     VISITOR         VISITOR
```

These phases are implemented via the *Visitor Pattern* which is a good pattern for traversing trees like the AST produced by our parser. A new interface, ``, has been introduced:



```go
type Visitor interface {
    Visit(node *ASTNode)
}
```

The Visitors that implement these phases both derive from a common visitor algorithm that is implemented by a new type called `DispatchVisitor`:

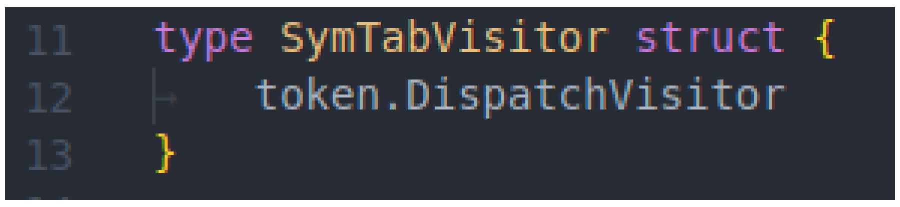


```go
type Visit func(node *ASTNode)

// A a general-purpose Visitor implementation that can be customized via the
// Dispatch table provided to it. This type is made to be embedded into Visitor
// implementations
type DispatchVisitor struct {
    Dispatch map[Kind]Visit
}

// Default visit function performs a lookup in the dispatch map, if no action is
// present, then this is a noop
func (t *DispatchVisitor) Visit(node *ASTNode) {
    if act, ok := t.Dispatch[node.Type]; ok {
        act(node)
    }
}
```

The `DispatchVisitor` implements the `Visitor.Visit()` method as a map lookup. Specialized Visitors can extend the `DispatchVisitor` by populating its `Dispatch` map, which defines semantic actions that should be taken upon discovery of a specific type of `ASTNode`.

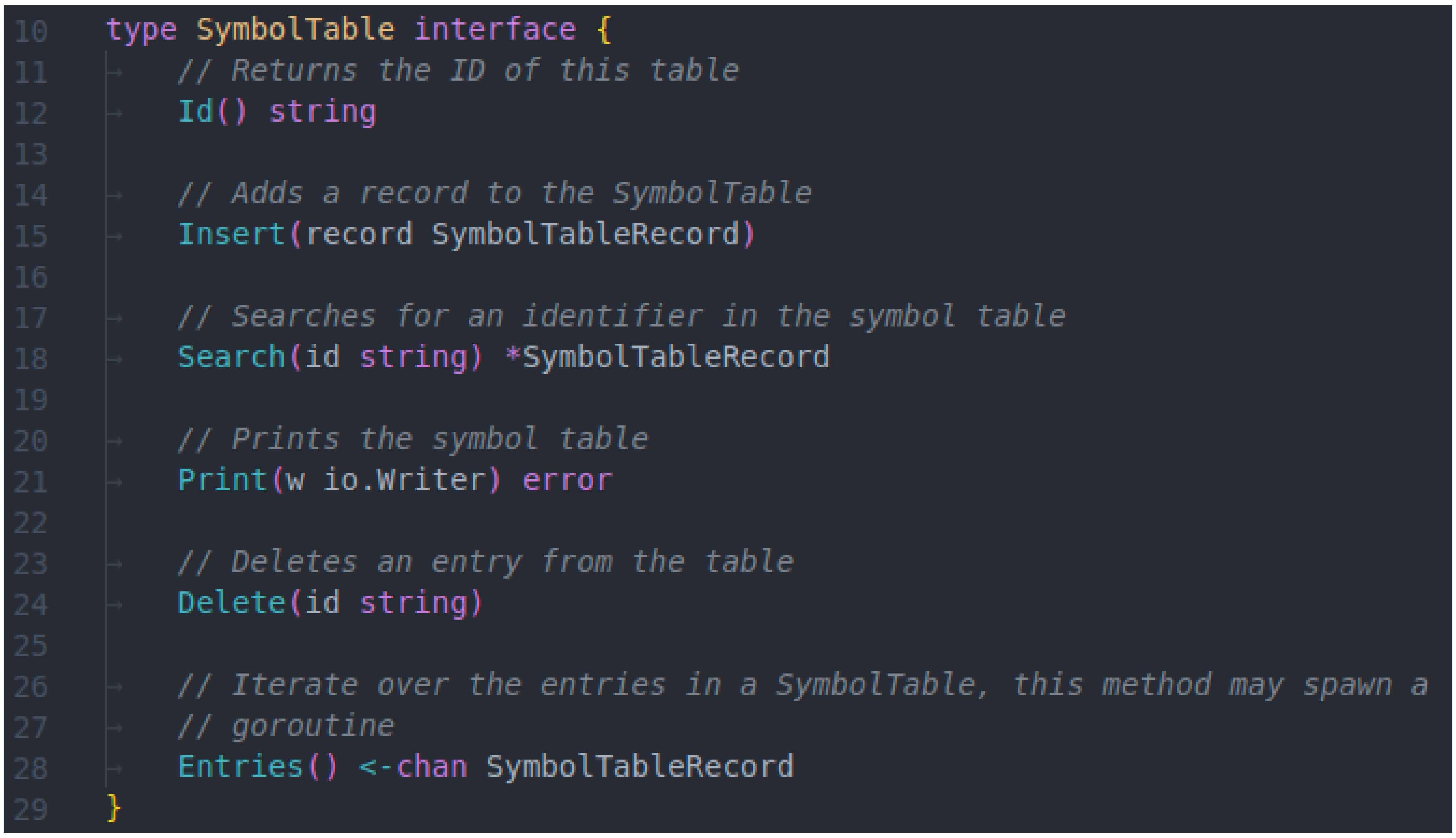
PHASE 1 - *Symbol Table Creation Phase*

The ***Symbol Table Creation Phase*** phase is accomplished by extended the `DispatchVisitor` through a new type called `SymTabVisitor`:



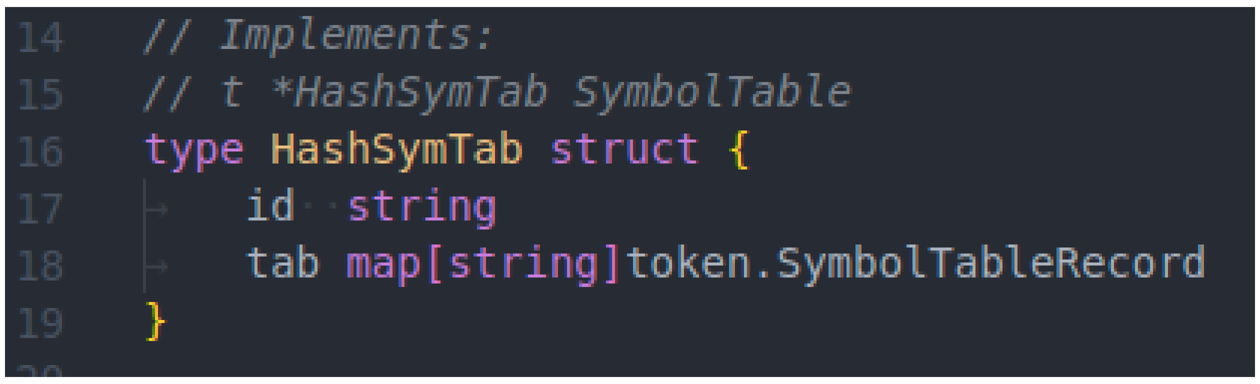
```go
type SymTabVisitor struct {
    token.DispatchVisitor
}
```

Here we are extending the Visitor through Go's built-in "embedded struct" mechanism. The only thing that `SymTabVisitor` needs to tweak is to populate the `Dispatch` map of the underlying `DispatchVisitor`. The Visitor fills this map with semantic actions that annotate the AST with `SymbolTable` instances. The `SymbolTable` interface defines methods for manipulating and searching a symbol table:



```go
type SymbolTable interface {
    // Returns the ID of this table
    Id() string

    // Adds a record to the SymbolTable
    Insert(record SymbolTableRecord)

    // Searches for an identifier in the symbol table
    Search(id string) *SymbolTableRecord

    // Prints the symbol table
    Print(w io.Writer) error

    // Deletes an entry from the table
    Delete(id string)

    // Iterate over the entries in a SymbolTable, this method may spawn a
    // goroutine
    Entries() <-chan SymbolTableRecord
}
```

It has a single concrete implementation in the form of the `HashSymTab`:



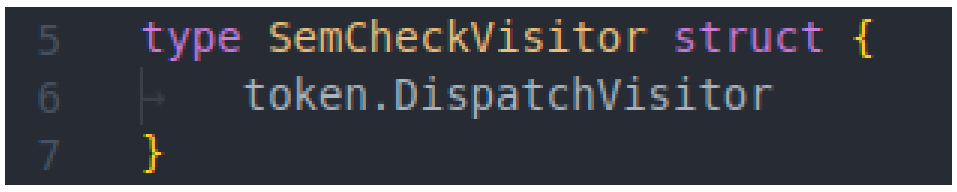
```go
// Implements:
// t *HashSymTab SymbolTable
type HashSymTab struct {
    id  string
    tab map[string]token.SymbolTableRecord
}
```

`SymbolTableRecords` may contain links to further symbol tables, which are defined on identifiers within the scope of the current symbol table.

The `SymTabVisitor` can be injected with a Go channel (concurrent read/write queue) upon which it will emit any errors and warnings encountered during the AST traversal.

PHASE 2 - *Semantic Checking Phase*

The ***Semantic Checking Phase*** is implemented by the `SemCheckVisitor`:



```go
type SemCheckVisitor struct {
	token.DispatchVisitor
}
```

This phase works basically the same as the ***Symbol Table Creation Phase*** described above. The `SemCheckVisitor` can be injected with a Go channel for emitting errors and warnings, just like the `SymTabVisitor`.

3 - Use of Tools

- `tool.go`: written by me for this project. This is a partial parser-generator that generates a large portion of the code used for the parser. Specifically, it creates the parse table, first, and follow sets, as well as code and function stubs used for the semantic actions of our attribute grammar.