COMP 520 : Compiler Design

Milestone 2 - Group 06

Design Decisions and Contributions

Scoping rules

Like in the assignment, we implemented a cactus stack of hashtables to implement the symbol table. At first, we thought it would be hard to differentiate between variables and types, but since GoLite forbids same-name declaration (`type a int`, then `var a a`), this became a bit easier to handle. We handle type-checking as a separate step from symbol table creation.

Upon any declaration where appropriate (type, variable, function), a special call `getSymbolScope` (identical to getSymbol, but only searches in the current scope and not any parent scopes) to see if the variable has been declared already. If not, then they are added to the symbol table and stored inside a new corresponding SYMBOL* field in the node of the AST as well.

Upon any access or use of an identifier, it's checked first if it exists inside the current or any parent scope with a call to getSymbol. Since type checking is done separately, we do not compare or infer any types at this point.

We change scopes when we encounter a blockStmt. We create a new symbol table and point to the scope we're leaving from as it's parent. This happens when we see function declarations, for loops, if statements, or switch statements.

User-defined types and nested structures were tricky to implement. Because our symbol table maps identifiers (char*) with a type, there is no easy way to cover user-defined types. We decided to make it a char* <-> char* mapping; we store the type of the symbol as a

string inside the symbol table. User defined types are therefore easy to put inside the symbol table. A type myType int statement would map the string “myType” to a type “int” in the symbol table, and any var declaration with type “myType” would map that variable name to “myType” in the symbol table.

Note: At the time of writing this report, we realize that this will incorrectly cause “myType” to be resolved to a variable of type int. we believe to fix this, we should have two different types of Symbol Tables: one that stores type information, and one that stores variable information.

To cover nested structures, we had a helper function called getType which took in a TYPE* struct and outputted in a pretty format its structure. For example,

```
struct{
            a int
            b int
}
```

would be stored as “struct{ a int, b int }”, and

```
var a [3][4]float64
```

would be stored in the “kind” (aka type) field as “[3][4]float64”. In the end, this makes type-checking a bit simpler, since we only need to perform a strcmp to make sure types match. In fact, this also allows us to easily check if a comparison between the following two structs is valid or not (this is valid):

```
var q struct{
      a int
      b bool
}
var p struct{
      a int
      b bool
}
```

Since there is no assignment compatibility (aka int + float operations being valid), then the way that the types are stored should be fine.

Function calls are hard because they follow identical syntax with typecasting. In the case that there is a function definition with the same name as a type name, then the symbol table creation phase will reject the program. But during typechecking, it's not required to declare a "type-cast" function before using it. Therefore, when we have function calls, we also check the type symbol table, to see if it is a proper typecast.

(this is not yet implemented because we only realized this at the time of writing the report.)

Type Checking

Typechecking was particularly difficult, not just because of the volume of things to take into account but also because it made us question some design decisions we made in the parser and tree parts. We had to work with what we had and realized some decisions could have been made differently (especially for complex data structures such as arrays, slices, and structs). While trying to implement array and struct comparison for in binary expression operations that requires two comparable argument types, we realized that the implementation of our expression node cannot support that. In fact, we stored all their different properties but provided no way to access them for typechecking. Because of that, we are currently unable to resolve the type of arrays, slices, and structs. A second issue of the same kind that we encountered is that we did not provide access to the return type of functions as well as the type of each struct field, i.e. we cannot typecheck function calls and struct field selection correctly. One discrepancy that we found in the specifications and the reference compiler is that the function name is an expression and we implemented it as an identifier. Each of the above mentioned implementation issues are described in details in the comments of type_checker.c. We also found out that some parts of our tree was not well linked together or did not make as much sense as we initially thought (at least for typechecking) but we still work through it. There were also surprises about things that were allowed or specified further in the specifications that we did not completely take into account for in our design. Particularly how expansively expressions were allowed in the GoLite code structure.

Invalid programs

The descriptions of the type-checking or symbol-table-creation problems of each invalid program we wrote is detailed briefly at the top of each file as a comment. We essentially tested that functions returned the proper types, that there are no duplicate var declarations, that short declarations override previous declarations iff there is at least one variable on the LHS that is undeclared, etc. We tested that user defined types behaved properly, and we tested scope rules for if/else statements, like in the scope3.go file.

Contributions

Since we had a lot to fix from Milestone 1, a lot of our time was spent on that. Ting was responsible for fixing all the errors from the parser & adding some weeding related to the previous milestone. All three of us worked on and fixed pretty printing together, verifying that our AST was properly built.

For this milestone, since we spent a lot fixing up our previous errors, we didn't have much time left to work on this. Ting was responsible for the symbol table and worked on the symbol table, Simon worked on the initial symbol table and worked on the typechecker with Su, and Su was responsible for the typechecker and worked on the typechecker. Simon and Ting wrote the invalid test cases for the first section. Note that we may have a bit over 30 test cases but that's because we wrote them separately. Since the different parts are all coupled together we had all to touch a bit of everything too.