MPCS 51300 Compilers final report

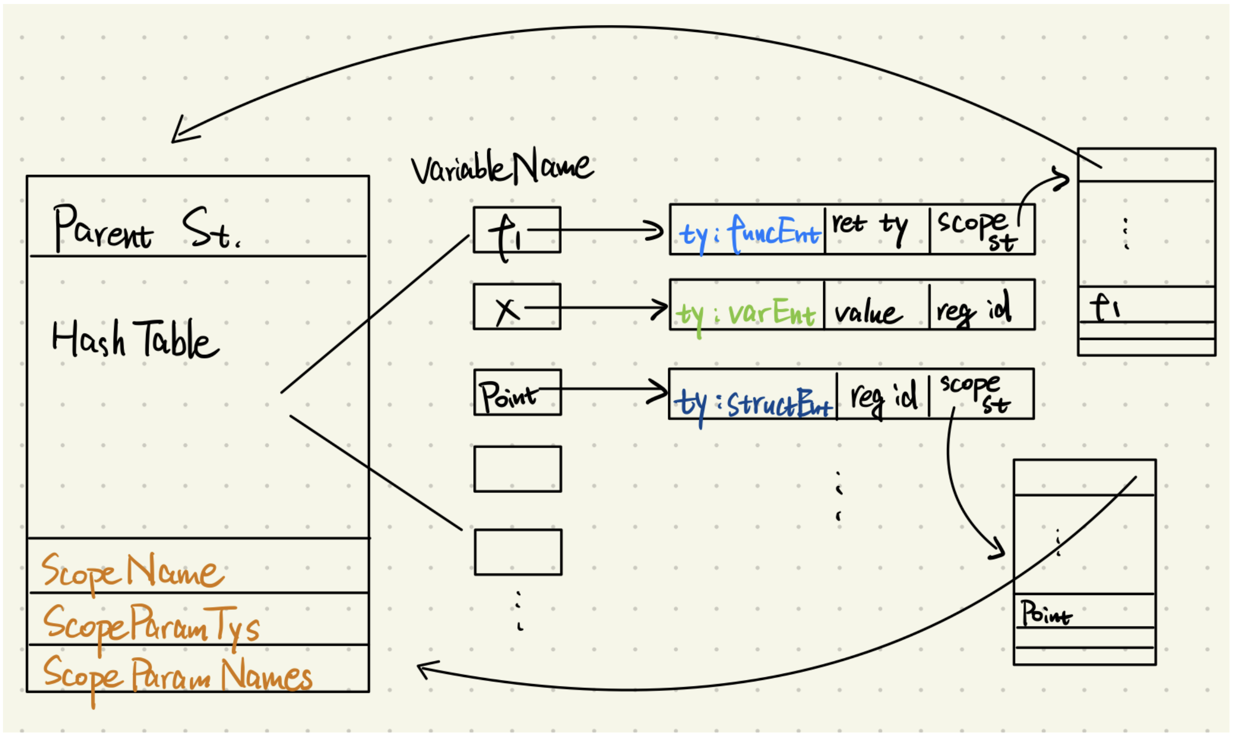
Autumn 2021
Yuanheng Zhao

In this quarter we walked through the structure and the workflow of a compiler: Starting from the front end, which handled by the scanner tokenizing the input source file and the parser building an abstract syntax tree and performing semantic analysis, we then practiced translating the AST to an intermediate representation, iloc, and finally went through to the back end, translating the ILOC to ARM code that could be identified and built by the server. In most of the parts, we followed the specifications and instructions taught in classes or in class materials, and I would like to introduce and discuss different phases of our GoLite compiler with you.

Scanner (golite/scanner) is the first component of our compiler. We applied the same strategy of the scanner as for homework 2. That is, we put a reader object into the scanner struct and would try reading characters until it breaks the possibilities of constituting any valid tokens (and we would rollback one character) or reaching the end of file. By applying three types of regular expressions, we got to identify the numbers, identifiers, and white spaces from the input. And we built two two maps, “keywords” and “symbols”, to distinguish keywords such as “let” and “Println” from the identifiers and retrieved the symbols such as “+” and “&&”. Additionally, we took care of line numbers by recording in the scanner object and appending it as a field of any tokens, and ignoring comments by setting a boolean variable isComment to indicate whether the whole line is comment.

For the parser part, we parsed and built the abstract syntax tree by applying a recursive descent parser on nodes of different types but implementing the same interface. All the nodes implement a base Node interface, which has TokenLiteral, String, TypeCheck, and TranslateToILoc as its function signatures. There exist three interfaces inherited from Node, Expr, Stmt, and Func, which help to distinguish categories of nodes and play corresponding roles in subsequent steps. The part responsible for parsing the tokens into an AST is in the file golite/parser/rdp.go, where we strictly followed the GoLite grammar rules specified in the language overview page.

With the AST built, we then do semantic analysis by implementing PerformSABuild and TypeCheck on the nodes of the tree deeper and deeper. During this phase, building and checking a symbol table is the crucial part, and the symbol table will also play an important role in later translating. Thus, we designed the structure of the symbol table to make it keep information that would help us in the future. There exist three part of our symbol table: the pointer linking to the parent symbol table, the hash table holding all declared elements (variables, structs, and functions) inside the current scope, and the fields storing necessary of the current scope (scope name, scope parameter types, and scope parameter names). Inside the hash table, we could grab an entry of any variable name existing in it. There exist three categories of entries, respectively for functions, structs, and regular variables (int and bool), for which we used to keep track of what the identifier represents, as shown in the following picture. For example, if there exist a global function named f_1, we could find a key - entry pair in the hash table of the "meta" symbol table whose scope name is "global", and by checking its entry we could find out the type of it (a function type), the return type of it, and a pointer linking to its scope symbol table. Why do we want it to have a scope symbol table here? Since a function may have arguments (parameters) prompted and its own fields, we would like to store those pieces of information as well as distinguish them with the scope symbol table that declares or invokes the function. Additionally, the scope symbol table of f_1 would point back to the current table as its parent symbol table.



We made PerformSABuild as a function signature of the Stmt interface we mentioned before. We performed building the symbol table by traversing from the root node (the program node) and calling PerformSABuild of each statement node encountered. When we encountered a newly declared variable/function we checked whether there had been an item with the same name existing in the current symbol table and constructed and added the variable and entry pairs into the table if it wasn't redeclared, and we also checked if there exists any invoking of undeclared variables or functions. One crucial point to notice is that the function (PerformSABuild) will switch to the scope symbol table of a function or struct when it goes deeper into type declarations or functions, and so that declare and check variables in the correct scope. With no errors happening in building the symbol table, we were able to perform Type Checking from the root node. While only statement nodes have function signature of PerformSABuild, all the nodes inheriting from the base node interface have function signature of

TypeChecking. During the phase of type checking, we set the type of entries of variables by scanning the indicator of declaration (e.g. var x int;) or looking at the symbol table and ancestor symbol table (check if it is a struct). We achieved this by applying the method taught in classes, constructing type signaltons. And we checked the consistency of types when we encountered assignment (left type must be the same as that of the right expression), boolterm (both the left and right terms must be bool if there exists any right term(s)), etc, and some other places requiring specific types, such as the type of the expression after an “if” inside a conditional statement must be bool.

We produced ILOC by calling the TranslateToILoc we implemented at the root of the AST. Following instructions in videos and classes, we implemented the interface and structure in the ir folder (golite/ir), which enabled us to produce instructions neatly, and we also revised and added specific fields into some different types of instructions to make it easier for the step of code generation, such as an integer indicating the register of the variable in Read. Based on the iloc rules we learned, we translated most nodes by first translating its children and then translated it by creating new instruction structs with target register id and source and necessary contents passed in. In this phase, we assign infinitely increasing ids of registers to the target by calling NewRegister in golite/ir/generators.go. We store and pass the instructions as a slice of strings, and append instructions to the end of it when translating new nodes, in the same way as we passed errors in performing SA build and type checking. And the slice of instructions inside a function was stored as the body of the function frag. The final results of ILOC would be a slice of FuncFrag, holding its function label and slice of instructions as its body.

For the code generation part, we added a function TranslateToAssembly into the interface of instruction in the ir folder (golite/ir). To translate a single instruction into ARM code, we invoke its TranslateToAssembly and will get a slice of string representing the ARM code. Since every type of instruction has its own rule, we implemented the function based on the slides and examples provided. The driven file is golite/arm/iloc2Assembly.go, which prompts for a slice of function frags and the global symbol table, and it will use the symbol table to retrieve necessary information such as the register id inside the entry of a parameter (retrieved from searching the entry of the parameter by its name from a scope symbol table of a function). Additionally, we created a map funcVarDict to map the register ids (infinitely increasing assigned in ILOC phase) to offsets for later usage of load and store, and a map paramRegIds specifically for mapping register ids of parameters in ILOC phase to default register ids (x0, x1, x2, …). Unlike providing infinitely increasing register ids as in ILOC phase, we created a map which stores and allocates available registers inside golite/utility/reg_allocation.go, and frequently occupied, released, and retrieved the next available register during this phase.

Working on this project the whole quarter with my partner, I got a better understanding of the concepts learned in classes. Like solving a puzzle step by step, we feel excited about assembling different parts from the front end to the back end together, even though sometimes it was pretty hard to think of a solution and it was time-consuming. We are proud of finishing our first compiler!