Jonathan Roven
Olivia Jerdee

Compiler Paper

Lexing and Parsing

Our compiler lexes a Golite program by identifying strings and characters such as keywords and operators with ANTLR. The lexer assigns all of the program into different tokens. If there are any parts of the code that don't match any of the tokens, our compiler returns lexer errors. Next, our compiler parses the Golite program by first having a grammar of regular expressions which is taken by ANTLR to create a parse tree.

Once this parse tree is created by ANTLR, our compiler traverses it to create an Abstract Syntax Tree. We do this by implementing the listener functions created by ANTLR so that we can walk through the parse tree and return our own AST with the root being the program. The exit functions work by generating AST nodes based on the tokens present in the expressions. We defined all of these AST nodes in the ast directory, and added a String method for each node so that we can bring the entire AST if the -ast flag is used. If any of the code doesn't match the expected grammar of Golite our compiler returns syntax errors.

Static Semantics

Our compiler implements static type checking by first traversing through the AST nodes and building a symbol table that stores all the global variables, struct declarations, and function information used in the program. This symbol table is useful for the rest of the compiler's steps and is used immediately in static type checking to verify that the types used in AST nodes are appropriate. Each node has a TypeCheck() function which is checked when the AST is traversed through. For example, in an assignment node it is checked if both the lvalue and expression on the right of the assignment have the same type. Otherwise, a semantic error is returned. The symbol table allows the compiler to see what the types of struct fields, local function variables, and global variables are. We also made GetType() functions for AST nodes, such as an intlit returning an integer, so that the compiler can type check all of the AST nodes.

To check whether all control-flow paths have a return statement, we implemented a ReturnCheck function that is called on each function node and returns a compiler error, which is nil if all control-flow paths have a return statement. This function traverses the statements in the function in reverse order, with recursive calls to the preceding statement if the check has not been satisfied. We added a ReturnCheck function to the statement interface returning a bool representing whether the statement itself satisfies the return check. It returns true for return statements and false for most other statements. For conditional and loop, we also added a ReturnCheck function for blocks, which returns true if a single statement within the block satisfies the return check. Conditional returns true if and only if both the true block and the false block return true, and loop returns true if the body block of the loop returns true. All nonlinear control flow is contained within conditional and loop statements, so ReturnCheck can traverse each statement linearly in reverse. As soon as it finds a statement that returns true, ReturnCheck immediately returns nil. If it gets all the way to the beginning of the function and sees that there are no more statements, ReturnCheck returns a compiler error.

LLVM Intermediate Representation

To implement LLVM instructions we defined a TranslateToLLVM() function for each AST node. We then got the LLVM instructions by traversing through the AST as needed and storing the LLVM instructions within an LLVMProgram struct. To write the instructions into an ll file, we added a String method for the LLVMProgram, each LLVMFunction, and every type of LLVMInstruction.

One of the challenging aspects of implementing LLVM was control flow. We created a cfg package and defined three functions to create new basic blocks: NewBlock (called only once at the beginning of each function), NewIfBlock (called at the beginning of conditionals), and NewForBlock (called at the beginning of loops). We decided not to have an exit block for each function as it complicated the translation of return statements in niche situations with complex control flow. Jumps to other basic blocks are implemented with ICMP instructions followed by breaks. We often had to add trunc and select instructions to make sure that the types matched, due to the decision to implement bools as i64.

Optimizations

The optimization that we implemented was dead-code elimination. Consider each function as a control flow graph with LLVM instructions. The final instruction of each basic block must be either a break or a return. Both of these possibilities leave the current basic block, either jumping to another or returning. Therefore, if there are more instructions after a break or return in a basic block, these instructions are unreachable. If the optimization is enabled, unreachable instructions after break or return instructions are deleted.

One challenge that this presented was dealing with basic blocks with no instructions. Any LLVM representation that we generated that had a basic block with no instructions did not run and instead gave an error. This problem presented some difficulty for us at first because it was not representative of any actual errors in our code, but simply a quirk of the nature of LLVM blocks. All blocks in question ended up not having any break statements that jumped to them, so it was simply a matter of getting around the error generated by the lack of instructions. With optimization disabled, the fix that we came up with was adding a single break instruction that jumps to the block itself. This solution was not ideal because it created unnecessary instructions and theoretically introduced an infinite loop, even though the loop was unreachable. With optimization enabled, these empty blocks are simply removed from the program.

We also implemented a version of dead-code elimination in assembly, albeit on a smaller scale. Once we had rewritten the register allocation scheme from the naive stack-based scheme to linear scan, we noticed that a lot of variables in succession had a live interval of length one. Due to the nature of linear scan, these variables were assigned to the same register. We started to see a lot of consecutive mov instructions where the operand and register were the same register. These instructions were the result of LLVM store instructions involving one or more temporary variables that immediately went out of scope. While they are necessary at the intermediate representation level, these instructions become obsolete in assembly, because both sides of the store instruction are the same register. The optimization removes any mov instructions with identical operands and registers.

One other instance of our assembly dead-code elimination is the removal of b instructions that jump to the next label and don't skip any instructions in between. These unnecessary instructions are related to the nature of LLVM, where every basic block must end with a break or return, even if the break instruction jumps to the very next basic block. This property is not true for assembly, so the optimization checks whether the next label is identical to the destination in the b instruction, and removes the instruction if this is the case. The situation becomes slightly more complicated when the b instruction has a condition code attached, because our assembly translation generates two b instructions, one for the true block and one for the false block. At most one of these instructions can jump to the very next label, so we check if there is a label within the two instructions immediately following the instruction for each instruction, and remove it if it exists and matches the destination.

Code Generation and Register Allocation

To translate LLVM to assembly, we created a TranslateToAssembly function for each type LLVM instruction. In some cases, the literal translation from LLVM to ARM was not too different, but the main challenge was to store all of the values in the ARM code which is even lower than LLVM and thus can be quite verbose. We started implementing ARM with a naive stack approach, where first generated address spaces for each LLVM register then stored the values within those address spaces. We accessed the LLVM registers within a function using the symbol table. The issue with this naive stack based implementation is that all of the loading and storing done by the ARM program is costly in both memory and time and is often unnecessary. To remedy this problem, we implemented a linear scan algorithm for register allocation to store many of the variables in registers instead of on the stack. The code for this algorithm exists in llvmFunction.go. We start by calculating the live range of each variable to ensure that no two overlapping variables are assigned to the same register. Then we use the live ranges to assign each variable its own register, spilling onto the stack if the number of variables in the same scope is ever greater than the number of available registers. The resulting register and stack address assignments are stored in the ARMFunction.

Another interesting aspect of the translation from LLVM to assembly is dealing with global variables. Global variables are not given a location in the map by the ARM program before running but are instead given an address on the stack. Thus, global variables have to be loaded in or stored differently than local variables.

Miscellaneous

Implementing the translation to ARM was one of the hardest parts of the compiler. Assembly code is much harder to read than LLVM, which made understanding and debugging it a lot more difficult. A major struggle was the realization that we would have to reimplement most ARM translation functions after we switched from the naive stack-based allocation scheme to linear scan. Most LLVM instructions, particularly load and store, proved to be extremely complicated. Each LLVM instruction translated into one or more ARM instructions that depended on whether the LLVMOperand and LLVMRegister were already in ARMRegisters, stored in the stack in ARMAddresses, or simply immediate values. Out of all the aspects of ARM translation, we spent the most time on sorting out these combinations and the ramifications that came with changing each case.

Overall, we really enjoyed building this compiler and learned a lot in the process. It was very rewarding to understand how compilers work under the hood, and even more rewarding to have the opportunity to implement one ourselves. We both feel that we now have a much better understanding of the steps taken to compile a program. Even though it was tedious at times, we were very excited to learn about assembly language, as it had always seemed so daunting in the past.

Division of Labor

We both collaborated evenly to implement the AST, LLVM, and ARM. We also collaborated on most other aspects of the compiler, but usually one of us focused on each aspect more than the other. Olivia mostly implemented the lexer, parser, and semantic analysis. Jonathan mostly implemented control flow, register allocation, and optimization. Across both LLVM and ARM, Jonathan focused mostly on control flow and binary expressions, while Olivia focused mostly on fields and globals.