Compilers Final Paper
MPCS 51300

Annie Huang and Kevin Lim

March 2025

1 Introduction

Here is a brief overview of the structure of this paper. We will walk through the phases of the compiler chronologically, starting with the lexing/parsing phase, then the static semantics phase, then the LLVM IR phase, and finally the code generation and register allocation phase. In each section, we discuss the main goal of that phase, how that goal is accomplished, and the important data structures used in the process. There will also be a brief discussion on the individual contributions of each group member, though the work was split quite evenly throughout. At the end of the paper, there will also be a section dedicated to how we might have added optimizations to our compiler. Thank you to Lamont Samuels for making this a great learning experience! < 3

2 Lexing and Parsing

The lexing and parsing phase is the first interface between a program and the compiler. As such, this phase is responsible for interpreting what a human has written and translating it into a form which is more amenable to future analysis. This phase also ensures that a program abides by the basic rules of the language — specifically, it performs *lexical* and *syntactical* analysis. If we are to draw an analogy to the English language, these analyses would be akin to verifying that a collection of letters forms words, and that the resulting collection of words forms sentences.

2.1 Lexing

The lexer is responsible for delimiting a string of characters into discrete groups which each hold a singular meaning within the context of the language. This process is called *tokenization*, and the resulting groups of characters are called *tokens*. We accomplish this by using ANTLR (ANother Tool for Language Recognition), which is a code generation tool specializing in the lexing and parsing of languages. We first supply ANTLR with a textfile describing the different tokens of the language using regular expressions, and ANTLR generates code which is able to process a string of characters into tokens (as well as catch any lexical errors). Under the hood, what ANTLR is doing is translating our textfile into a *Finite Automata* (FA), which is essentially a finite state-transitioning machine which evolves as a function of a discrete stream of inputs. By using the ANTLR-generated code in our compiler, we are able to easily translate a given program into its corresponding collection of tokens.

2.2 Parsing

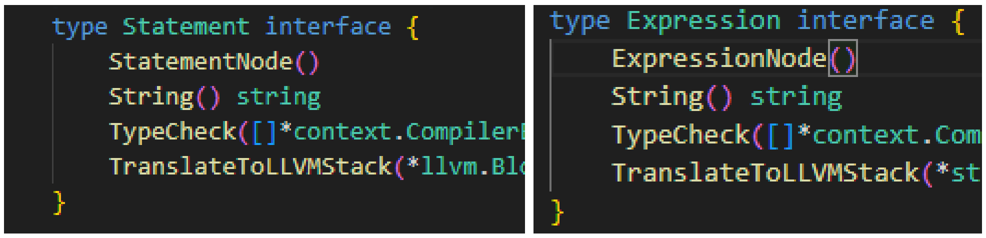
The parser is responsible for verifying that this collection of tokens is syntactically valid. As a byproduct, it also creates a parse tree, which better represents the syntactic relationships between tokens. The parse tree is generated by adding edges between tokens that depend on each other syntactically — for example, the meaning of a token representing an assignment depends upon other tokens denoting the value being assigned and the alias to which it is assigned. Thus, by creating the parse tree we are also confirming that the meaning of the program could be completely derived from the supplied tokens. This phase is accomplished again by using ANTLR — we first supply ANTLR with a *grammar*, which is an exhaustive specification of all the sentence types our language supports. ANTLR then provides us with code which is able to process a stream of tokens and generate the parse tree, as well as catch any syntactical errors along the way. While this process is similar to the lexing phase, a key difference between these analyses is that a grammar may be (and often is) defined recursively — as such, constructing a FA for parsing is unviable. ANTLR instead uses a recursive derivation technique, where collections of terminal and non-terminal symbols are continually transformed until a desired form is achieved.

Annie did the lexing and Kevin did the parsing for this phase.

3 Semantic Analysis

3.1 Type Checking

In semantic analysis, the compiler verifies that the generated parse tree conforms to the static semantics of the language. More specifically, the compiler first translates the parse tree into an *Abstract Symbol Tree* (AST) — a reduced form of the parse tree which retains key pieces of information — before validating the semantics of the AST. The AST is generated by recursively walking the parse tree (DFS-style) and calling exit functions upon the exit of each node. By doing this, we ensure that for each node, the entire subtree rooted at that node is processed before the node itself is — this allows us to collapse the information of subtrees in the parse tree into singular nodes in the AST. Importantly, while the AST may be smaller than the parse tree, it retains the important relational information from the parse tree. Once the AST is generated, the compiler validates it by traversing the AST and performing type checking. Type checking is done recursively — we check the type of some root node which, in the process of checking its own type, will query the types of its children. This is done by implementing a TypeCheck() method for each node in the AST, which effectively walks the tree and verifies that all nodes have the correct type. Figure 1 shows the TypeCheck function signature in the Statement and Expression interfaces.



```
type Statement interface {
    StatementNode()
    String() string
    TypeCheck([]*context.Compiler
    TranslateToLLVMStack(*llvm.Bl
}
```

```
type Expression interface {
    ExpressionNode()
    String() string
    TypeCheck([]*context.Com
    TranslateToLLVMStack(*st
}
```

(a) Statement interface (b) Expression interface

Figure 1: Our implementations of the Statement and Expression interfaces for the AST. Essentially all AST node types fall into one of these two groups.

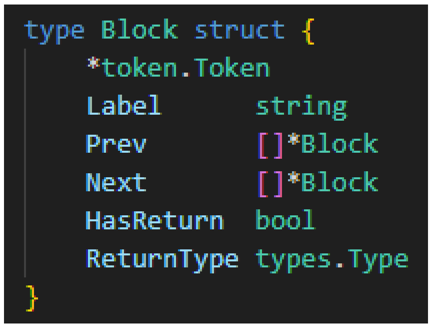
Semantic analysis also involves performing additional checks on the validity of certain aspects of the program — for example, verifying that a main function exists, that all field identifiers belong to their corresponding struct definition, etc. These checks are performed within the TypeCheck method.

3.2 Control Flow

Another important step in validating a program is ensuring that all control flow paths have a return statement of the proper type. Doing this effectively requires translating the AST into a new form which better describes the program's flow of execution — this form is called a *Control Flow Graph* (CFG). The CFG is creating during the type checking phase — in fact, in our implementation, the CFG is created as a byproduct of the TypeCheck methods. This is done by passing around pointers to structs representing blocks in our CFG during the recursive walk of the AST. At points of junction, such as entries to for loops and if statements, new blocks are created and passed down. Importantly, at return statements, certain fields are set to denote the presence and type of the return statement. Figure 2 shows the fields of a CFG block to better explain the information each block contains.

Once the CFG is created, it is validated by another DFS-style traversal. This traversal checks for two related aspects of the CFG — firstly, that all control flow paths in a function have a return statement, and secondly that all return statements in a function have the proper type. Since we can assume that the fields of each block in the CFG have been set properly at this point, performing the check is a simple task and can be done recursively.

Annie and Kevin both worked on the type checking and Kevin did the control flow validation for this phase.



```go
type Block struct {
    *token.Token
    Label      string
    Prev       []*Block
    Next       []*Block
    HasReturn  bool
    ReturnType types.Type
}
```

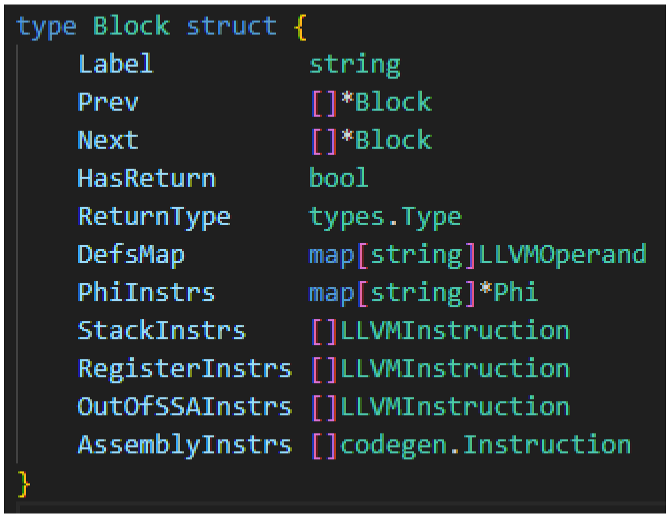
Figure 2: Struct definition for a block (node) in the CFG.

4 LLVM IR

This point marks the transition from the "front-end" to the "back-end" of the compiler. What this means is that the original program has been sufficiently validated, and that future phases will focus on optimization and the translation of our current representation into machine-readable *Intermediate Representations* (IRs). The first form we will translate into is LLVM, which is a type of high-level assembly language. More specifically, we will translate into a stack-based LLVM, which means that we will use the stack (as opposed to registers) to store data.

4.1 Stack-Based LLVM

The translation to the stack-based LLVM IR begins with another traversal of the AST. This is done to construct a new version of the CFG, this time with different nodes containing more information (I'm not sure why we did it this way it was probably unnecessary). Figure 3 shows the fields of this new CFG block.



```go
type Block struct {
    Label           string
    Prev            []*Block
    Next            []*Block
    HasReturn       bool
    ReturnType      types.Type
    DefsMap         map[string]LLVMOperand
    PhiInstrs       map[string]*Phi
    StackInstrs     []LLVMInstruction
    RegisterInstrs  []LLVMInstruction
    OutOfSSAInstrs  []LLVMInstruction
    AssemblyInstrs  []codegen.Instruction
}
```

Figure 3: Struct definition for a block in the new CFG.

An important distinction between the old block and this new block is that this new block has multiple fields for instructions. In this phase, the field we care about filling out is the StackInstrs field. Stack instructions are generated by traversing the AST and calling TranslateToLLVMStack(), which is shown in Figure 1 to be a method for both the Expression and Statement interfaces. This method generates LLVM instructions and appends them to the StackInstrs field in the corresponding block.

An important characteristic of this intermediate representation is that it is stack-based. This means that nearly all variables — including parameters, local variables, and structs — are stored explicitly on the stack. Although this makes this representation easy to generate, it is also incredibly inefficient, and this fact will motivate our future translations.

4.2 Register-Based LLVM

The next step is to translate the stack-based LLVM IR into a register-based LLVM IR. Abstractly, this means that data will be allowed to persist in registers independently without being stored on the stack. Effectively, this means that we can prune many store and load instructions from our stack-based LLVM IR. We accomplish this by processing our CFG blocks in order and maintaining a map of variable names to their values. This is the DefsMap field in Figure 3. What this allows us to do is substitute in values in instructions and prune any stack-based loads, stores, and allocations. This phase also deals with the creation of Phi nodes, which manage the assignment of values to variables from multiple control flow paths. Using Phi nodes allows us to maintain the property of Single Static Assignment (SSA), which makes this form generally easier to work with and more amenable to analysis and optimizations.

Annie and Kevin both worked on both the stack-based and register-based LLVM IR.

5 Code Generation and Register Allocation

The final phase of our compiler is the translation of LLVM into assembly. While LLVM and assembly share many commonalities (similar syntax, "closeness" to machine code, etc.), one important difference is that LLVM assumes access to an infinite number of virtual registers, and assembly does not. What this means is that our translation from LLVM into assembly must contend with the fact that computers sadly do not have unlimited fast memory and allocate space on the stack for "spilled" data. We must also contend with the fact that physical registers may be reused within different contexts throughout the lifetime of a program, and use the stack intelligently to allow data to persist without registers.

5.1 Register Allocation

The first step in this phase is to assign physical registers to each of the virtual registers used throughout the LLVM program. This is done using a linear scan allocation algorithm, which performs a linear scan of the program and greedily assigns physical registers based on a liveness analysis. More specifically, for each function we iterate through the program and maintain the *live range* of all variables: the distance between where it first and last appears in the program. We then sort all live ranges by their starting times and iterate through them in order, assigning physical registers to variables greedily. Once a register is assigned to a variable it is bound until the end of that variable's live range, and if we ever reach a point where there are no more physical registers left for a variable we allocate a *spill register* for it, which is in actuality a reserved location on the stack. This phase also converts any Phi instructions into an equivalent collection of move instructions, officially stripping our program of the SSA property.

5.2 Code Generation

The final step is to generate assembly code from our LLVM. At this point, our LLVM program consists of instructions which 1. faithfully utilize the same physical registers which assembly assumes access to, and 2. are very similar to assembly instructions. This means that the translation to assembly is fairly straightforward.

The main difficulty in this step is properly managing the stack. Stack management consists primarily of two tasks: allocating space on the stack for spill registers, and using the stack to preserve data through the changing of contexts (stack frames). Stack allocations are done manually by manipulating the stack pointer, and certain registers (caller-saver registers) must be saved to the stack when making function calls — this helps give the illusion that each function has full control over a portion of the register set. One particularly annoying bug was not saving all caller-saved registers when branching to printf, which would sometimes result in corrupted data upon return — it turns out that printf uses many more registers than expected under the hood.

Annie did the register allocation and Annie and Kevin both worked on the code generation.

6 Optimizations

If we had more time, one optimization we would like to have added is dead code elimination. Dead code elimination involves removing portions of the program which are guaranteed to have no impact on the final result. Formally, a statement is declared *dead* if its result is never read in future instructions. We make the distinction between the *definition* and the *use* of a variable — a variable being assigned a value in some instruction is *defined*, and in all other appearances it is being *used*. If a variable is defined without ever being used afterwards, then that definition can be considered dead and the instruction can be removed without impacting the final result of the program. This process of removing dead statements can be repeated over and over until some convergence criteria is reached, since each removal of dead code may change the “deadness” of other statements. Dead code elimination may also include the removal of instructions after return statements which are unreachable. In our implementation, this process would be done somewhere between the register-based LLVM IR and register allocation phases. Dead code analysis could have been done very similarly to our liveness analysis, but instead of tracking all appearances of each variable we would make a distinction between definitions and uses. Basically, we would perform a linear scan of our LLVM code blocks and keep track of the uses/definitions of each variable in a map. We could also keep track of which instructions come after a return statement — since instructions execute linearly within each code block, any instructions following a return statement in the same block are guaranteed to be unreachable. This would have allowed us to identify dead statements, which would then be removed from the code block and the process would repeat.