Paul Kummer

Dr. Hanku Lee

Operating Systems CSIS 455-01

09/06/2021

Summarization of Chapter 1.2 and 1.6 “Compilers: Principle, Techniques, & Tools”

1.2 The Structure of a Compiler

Compilers will take code written in a high-level language and turn it into machine code that will execute a program on that machine. The process of compiling can be broken up into two larger parts of analysis (front-end) and synthesis (back-end). The analysis part ensures that the code is syntactically correct and creates a symbol table and converts the code to an intermediate representation. This intermediate code and symbol table is passed on the synthesis portion of processing. The synthesis portion of compilation will create the machine executable code that is run on the computer.

The compilation process can be viewed as a series of phases that will continuously change the code from a higher-level language into something closer to machine code with each phase. There is an optional phase of optimization that will make the machine code more efficient.

Lexical Analysis

The first part of compilation is lexical analysis, commonly called scanning. This process reads over the characters from the source program and turns them into meaningful groups. Each group of characters is called a lexeme. The lexeme is then turned into a token, consisting of a token-name associated with an attribute-value. The attribute-values are stored in a symbol table that is used link the token-names with their values during the next phases.

Syntax Analysis

Once the symbol table has been populated, syntax analysis occurs, also known as parsing. During syntax analysis, the compiler will create a syntax tree to represent the operations of the original code but will instead use the tokens as an intermediate representation. The intermediate representation will use values such as an identifier and its position or primitive values, which could look like <id, 4> or +.

Semantic Analysis

Now that the original code is represented in a syntax tree with an intermediate representation, it can have its values type checked and verify that values are valid. In this phase, operands are checked to ensure that they are the same types. If one operand is a float and the other is an integer. The semantic analyzer will try to coerce the integer into a float. If a value is invalid, an error message will be displayed, and compilation will stop.

Intermediate Code Generation

The basis of how the high-level code should perform should be transformed enough after syntactic and semantic analysis to allow the original code to be transformed into something closer to what the machine can understand. This is exactly what intermediate code generation

does. Based on the syntax tree and its intermediate representations, the nodes are transformed into an intermediate code called three-address code (3AC). 3AC can consist of a maximum of two operands, addresses, being stored to one address. Also, the order of operations is preserved by operators of higher precedence being performed first. When the code is in 3AC, it closely represents an assembly language and can be converted easily.

Code Optimization

Optionally a code optimization phase can be run to attempt to make the code run more efficiently. Efficient can be defined in different ways. The intermediate code can be condensed into less lines of code or it could be transformed into lines of code that consume less energy. How it is optimized is up to the users' specifications or the compilers capabilities. For example, some high-level coding for a loop may check an index on each iteration, but the compiler could optimize the code to do pointer arithmetic and reduce the number of instructions needed for each loop and eliminated the index check.

Code Generation

The last phase of compilation is code generation. In this phase, the intermediate code is translated into the target language for a machine. All the instructions to move, load, store, add, etc. into registers are converted into machine code.

Symbol-Table Management

Symbol-Tables have a collection of all the variables used within a program. Each of these variables have their own scope, values, and types. All this information is stored in the symbol-table and should allow the information to be retrieved quickly.

The Grouping of Phases into Passes

When compilation happens, the phases of compilation are grouped together into passes. In a pass, lexical analysis, syntactical analysis, semantical analysis, and code generation could be combined into one pass. Then, if the user decided to, another pass could be done for code optimization. Finally, the code generation pass would create the target machine code. Different passes allow many combinations of front-ends with back-ends to work with many source languages to target machines.

Compiler-Construction Tools

To aid in making compilers, there are tools programmers can use to make the process easier, much like an IDE. There are parser generators, scanner generators, syntax-directed translation engines, code-generator generators, data-flow analysis engines, and compiler-construction toolkits. They are all used to make construction of a compiler easier and less time consuming.

1.6 Programming Language Basics

The Static/Dynamic Distinction

Every programming language has a scope, section that something can be accessed from, for variables and objects. When the scope can be determined upon compilation, the scope is static. With static scopes, a variables value can be determined by looking at the code and tracing the variables path through functions, methods, or operations. Languages like C and Java are

statically scoped. Dynamically scoped languages have variables that the value cannot be known until runtime. There could be a variable with many possible values, but it will only be known when the program executes.

Environments and States

After a program begins to run, the value of a variable depends on what environment the variable is in. The environment is the mapping of a name to a location/variable in memory. Once the environment is known, then the location/variable can be mapped to its value, known as a state mapping. Both mappings are typically dynamic, but they can be static such as a final global variable or values being defined as constants.

Static Scope and Block Structure

As stated previously, static scoping is the predominant method for name resolution. Within a statically scoped program, values for every variable can be derived before runtime. Many languages use block structures, typically beginning with "{" and ending with "}", to confine variables and objects to that block. Block structures can be nested too, allowing use of values from the outer blocks, but if the inner blocks have redeclared a same value, the inner blocks value will be used and the outer block will be unchanged.

Explicit Access Control

Some languages use keywords such as public, private, and protected to specify the extent of scopes to either allow everything to access its values (public), nothing to access its values but itself (private), or only members of a class to access its values (protected). This is like nesting of block structures. The outer blocks cannot change the inner loops similar to a class being private. The inner loops can change the outer loops like protected classes can access its parent's values. Furthermore, outer blocks can access the values of the root block like public classes.

Dynamic Scope

If a value cannot be completely known before runtime, a program is at least partially dynamically scoped. This can occur in C if a variable is defined in a macro that is dependent on other variables used throughout the program. Then whenever the value of that macro is called, it will dynamically change depending on what the other variables values are within the environment of which the macro was called. Objects of a class can also have values that are unknown until runtime, making them dynamic as well. There could be a person object with a method that retrieves the person's age, but until a person is created, that person's age is unknown. Once people objects are created, they could all have different return values for their age. Therefore, person.getAge() could return a variety of values.

Parameter Passing Mechanisms

In programs, methods and functions often require parameters to be passed into to them so they can perform some operation. There are two different ways to view the parameters of a method or function. There are the actual parameters, which are the values used when the method/function is called, and the formal parameters, which are used for defining the method/function. The values of the parameters can be either call-by-value, call-by-reference, or call-by-name.

Call-by-Value

If the value passed into the procedure is copied from the original value, then the procedure's parameters are call-by-value. This prevents the original values from being altered and is the most common way parameters are called. The confusion of pass-by-value and pas-by-reference occurs when pointers are used. When a pointer is passed in call-by-value, the value of the pointer is copied, which that copy will still point to the same address in memory as the original pointer. This can allow the changes outside the scope of the block. For example, if an array is passed as a parameter in Java, the location in memory that the array points to is copied into the new scope. Then if any index of the copied array is changed, the actual array outside of the method will be changed. Using an array as a parameter in Java will cause a reference to that array to be copied to a new variable in the scope of the method. This prevents all the values of an array from being copied. Java is always call-by-value but has work-arounds that make is appear as though it is not. When a pointer is copied, there will be two separate pointers that have the same values, which is a location in memory. If these two different pointers were dereferenced, they would be the exact same variable, making them appear identical.

Call-by-Reference

When a parameter uses only the same memory address of whatever is passed as a parameter, the parameter is pass-by-reference. Using call-by-reference will allow changes to values outside the scope of the method/function/procedure, since the actual parameter will have the same reference to a memory address as the formal parameter. Pass-by-reference is very similar to pass-by-pointer; however, passing-by-reference cannot be reassigned and only refers to one and only one object. Using a reference is like using that object itself, just with a different name. Pointers can be reassigned and set to null unlike a reference and are just variables that store addresses in memory.

Call-by-Name

Call-by-name is very infrequently used in modern programming. When it is used, a formal parameter and actual parameter act as if they are the same. The formal parameter acted as if it was a macro for the formal parameter.

Aliasing

Call-by-reference is essentially creating an alias. An alias is two variables that share the same value. If either one of the variables changes the value, the change will happen for the other variable too. In Java, an alias is created for objects when they are passed because of the nature of copying a pointer as a new variable. This causes great confusion among programmers who have worked extensively with C++ and see Java acting just like call-by-reference, even though it does not work that way.

Works Cited

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: Principles, Techniques, & Tools (2nd ed.). Pearson Education.