Project 4: Optimizer

Stephen Sanders, Jared Wermager, Austin Johns

CST-405

Grand Canyon University

**Introduction**

In this project, we focus on the optimization of code and the translation of code into machine language for kernel processing. The optimization of our compiler is crucial to the functionality of the entire language processing. Optimization readies code to be processed by the computer. An optimizer can be very useful because it can improve efficiency and remove errors from code input. Secondly, we’re focusing on turning this code input into machine language that can be processed by the computer for concrete implementation. This is the final step of the compiler, and it’s what executes commands given by the user. The code is processed in the MIPS machine language. Each of these additions to the compiler we have designed make it better to use and implements the commands of the input.

**Three Forms of Local Optimization**

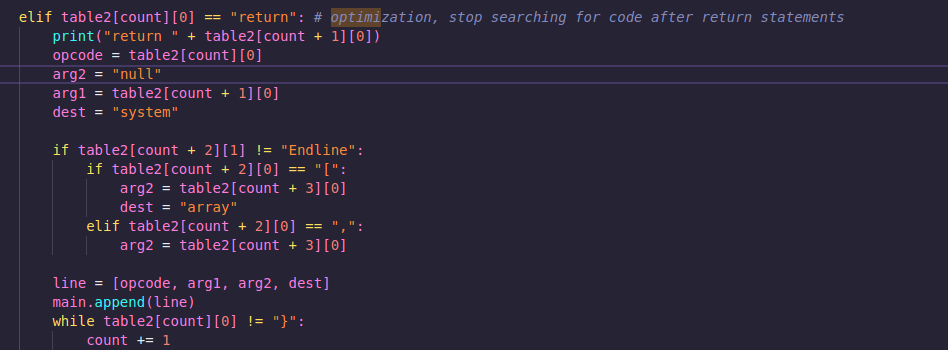
In this project and compilers in general, local optimization is important because this optimization helps ensure that all compilation of code is correct and as efficient as possible. For the compiler to interpret and execute code as the user intends it for the C-- code, the code must run through the lexer, syntax analyzer, and be optimized. The optimization of the code is the focus of the project, and is the final step in completing successful code interpretation. The first form of local optimization in this project is the focus on scope. When compiling a section of the code, the optimizer will not use future or past code that does not affect the current output. This is done through the use of the return statement in code. The return statement is the “stopping point” of the compiler. This optimization makes sure that all code is completed as it should be. The next form of local optimization is ignoring dead variables. These dead variables are unused claims of variables that exist in the code body. Even if a variable is changed or altered in the code or through a process, and the variable is never used, it is considered a dead variable. The optimization in the compiler helps free memory for a more organized and efficient data structure. The third form of local optimization is directed acyclic graphing, or topological ordering. The optimization gives tasks a clear ordering of completion. Similarly to a stack, but more flexible, the topological ordering allows for one task to begin, pause, and another task to start. This is important because if one task is dependent on another task, this task ordering is more flexible. The acyclic logic allows for every task to be completed in the necessary order.

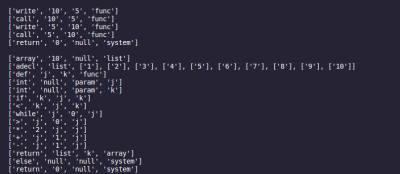
**Three Forms of Flow Optimization**

Flow optimizations are all about efficiency. The efficiency of our compiler is important because not all devices have expandable memory for inefficiency. These flow optimization changes help the optimizer handle data in a way that serves the user and the processor. The first optimization made with the compiler is the elimination of common subexpressions. This helps because the compiler has less computation to do. For example, if there's a computation of a + b and another computation of b + a, this is only calculated once, as both expressions yield the same result. Another great optimization of this compiler is the scheduling of tasks. This is done to ensure that not only every task is completed, but that every task has a time to be completed. This helps manage memory use. The final optimization of the compiler is the avoidance of unconditional jumps by code replication. This is crucial to the optimization of the code because if there is an unconditional jump in code processing, an entire process can be skipped.

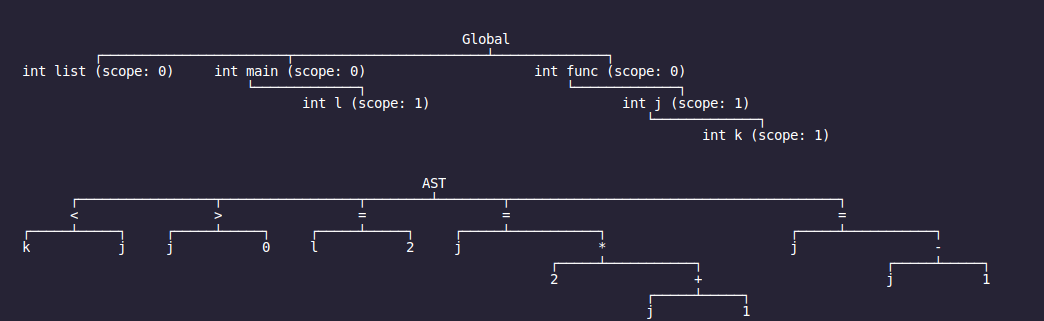
**Intermediate Representation**

Intermediate representation produces abstract machine language like three address code. This IR allows for one compiler instead of multiple which simplifies the process of developing a compiler. We decided to use three address code for our intermediate representation. Without IR and three address code, the front end of compiling would be very difficult, but our three address code allows for easier optimization and code generation.





**Symbol Tables**

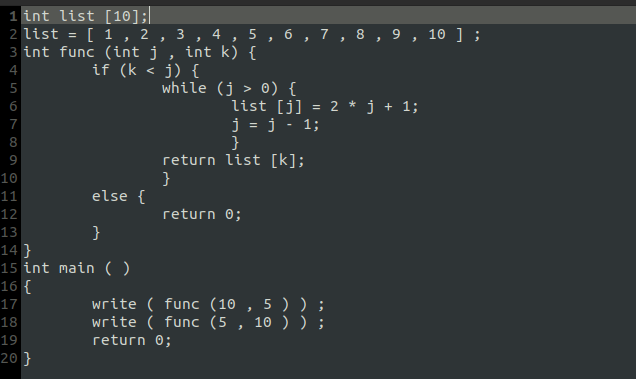


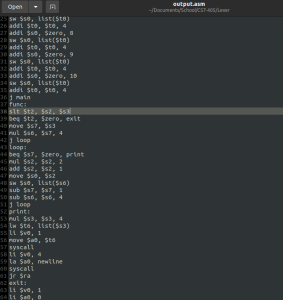
**Console Output in the Form of Machine Language**

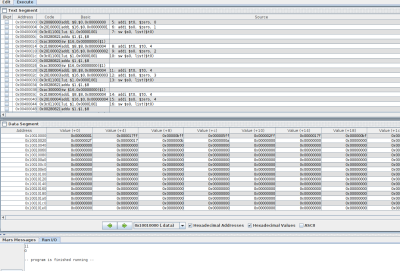
In the output of the code from our python-based C-- code optimization to the assembly machine language, there are many steps taken to ensure proper conversion of languages. In order for the machine language conversion to complete successfully, each token from the previous projects are converted into corresponding lines of assembly code via address code.

**Machine Language Running**

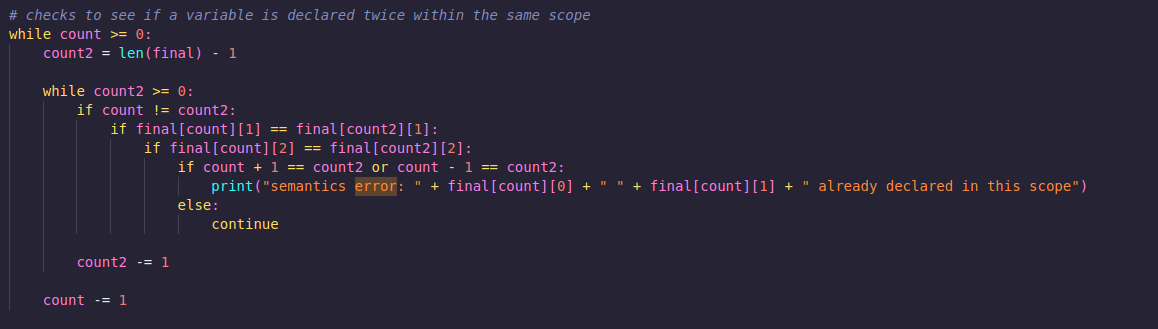
This project generates an output.asm file that is MIPS compatible. The generation translates the TAC shown below into definitive assembly code based on the parameters in each row of the the TAC. The example code outputs an 11 and a 0 when ran, which is definitive of printing out the return value of ‘list[k]’ and printing the return 0 value when k does not meet the requirements of the if statement. The array values after the while loop turn out to be, [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]. This turn of the program the k value was 5, giving us the output of list[k] = 11.

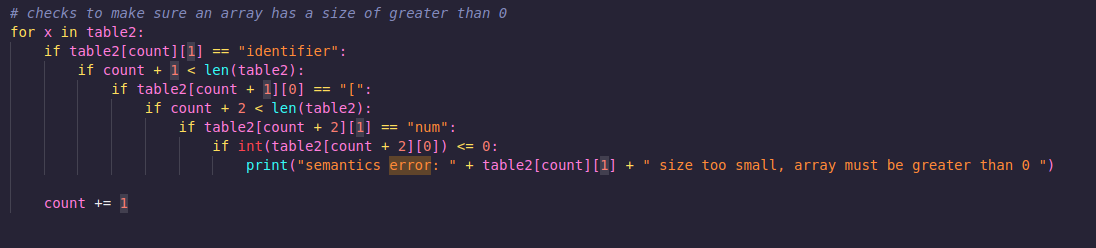






**Error Messages**





**Execution without Errors**

The execution of the program without errors simply prints the TAC to the console as well as creates/overwrites the ‘output.asm’ file with the assembly code generated. Most of the error checking takes place before the code generation stage of the compiler, meaning that errors we face are likely semantic or syntax related.

**Team Member Responsibilities**

*Code Implementation*

*MIPS Code*

*Three-Address Code*