Jason Fong

Brent Lee

CSC 431

Aaron Keen

**Final Paper**

**Front end**

Our compiler implementation includes a 32-bit architecture (everything includes integers, booleans, etc. are all represented in 32 bits).

For the front end of the compiler, a file written in the Mini language gets put into the lexical analyzer.

The parser, which is the next step, outputs the information in tokens as an Abstract Syntax Tree (AST). The static semantics that are used after the parsing, when given the AST, basically returns nothing, but prints out an error and exits if the static type-checking finds a type error.

The method we implemented is passing down the return type all the way through functions, and anytime a return statement is found, it compares the expected return type and the expression in the return statement. All functions require a return statement with the only exceptions of expecting a void type. If there are branches, the type checker had to pass the return type through the true body and the false body whether it be for conditional statements or while statements. The return type gets passed down to every part of the code.

The output of the entire AST then gets turned into a control flow graph, which is the intermediate representation before becoming assembly code and is also the transition to the back end of the compiler.

**Back end**

The purpose of the control flow graph is to “show the flow” of the program. It is a directed graph where each “node” (block) is set up to contain a set of instructions up until the flow is “disturbed” such as the case with a conditional statement and a while statement.

An intermediate representation (IR) is used in compilers so that the compiler systems, such as LLVM in our compiler’s case, can be used by different source languages to generate code for many different target architectures. Static single assignment (SSA Form or SSA) is a representation of the code such that each variable is assigned exactly once.

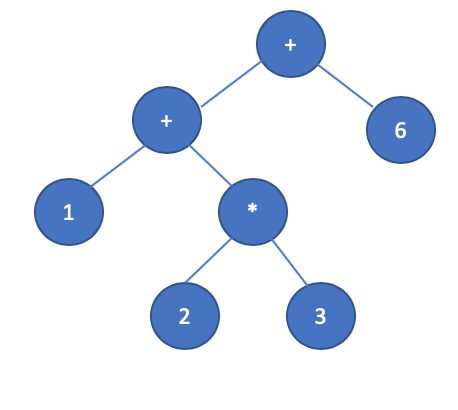
A benefit of SSA is that at the end of a conditional, where the “if block” ends and the “else block” ends, no matter what path it takes, a variable would only have to be using the same variable and a phi instruction, which checks which path the code took. Another benefit is the simplification of the IR to help make easier some optimizations.

We did not implement any optimizations for the compiler; however, we did implement the code generation and register allocation. For this to happen, we had to eliminate the phi instruction used in LLVM. To do this, we had to add a temporary register at the bottom of each previous block if there is a conditional or a break in the flow. That register was then used in the current block for assignment and use. The phi instruction had to be eliminated because ARM does not contain any phi instructions.

**Data Structures**

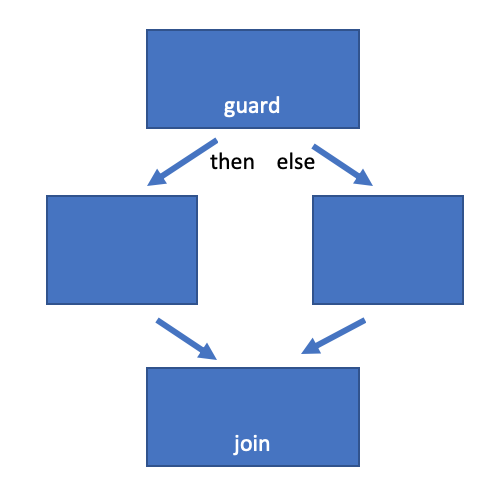
The first and biggest data structure used is a tree. We used a tree to represent the Abstract Syntax Tree and the order of which the code was parsed. The tree is read in so that it does depth-first search and is left associative so that binary operators can behave like they are supposed to. Here is an example

1 + 2 \* 3 + 6

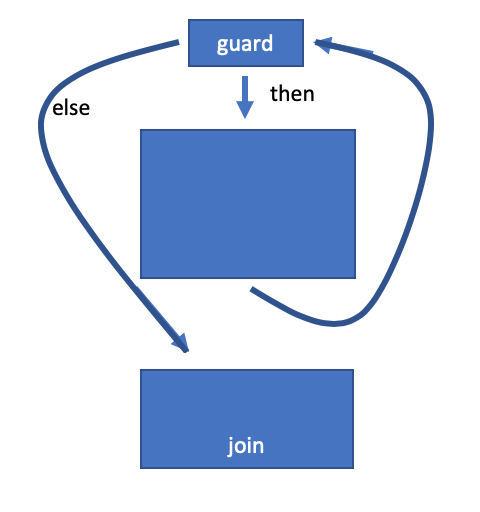


Another data structure used is a graph for the intermediate representation in LLVM. The graph, specifically the Control Flow Graph, is used to represent the flow of the graph. There are different ways to represent a loop, but we used one where the conditional guard is in a separate node, and the body is another node, and the exit is another node, as well. Here is an example of a conditional and a while loop as used in our compiler.

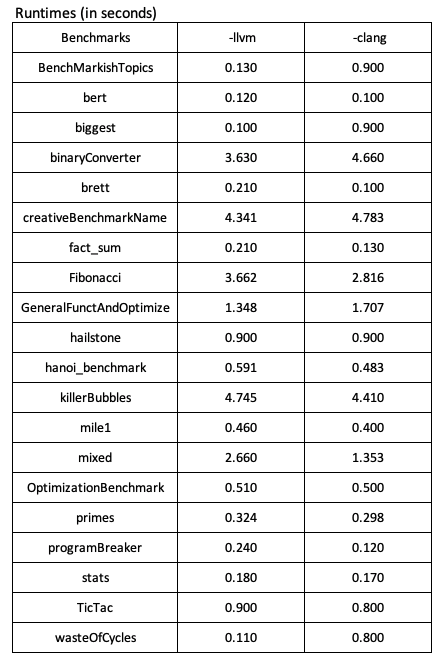
**Conditional CFG**



**While CFG**



**Performance Chart**



**Performance Review**

I think some of the times for clang were a little bit variable because it is on the server. Now is the time that CPE 357 is learning about pipes. I was testing the clang times as the servers were fork-bombed, so it started slowing down a little bit, and then it turned off. I went on all of the different servers Cal Poly offers and tested the compiler code on the fastest one.

Some of the LLVM code was a little faster than the clang one. I do not know necessarily how this happened, but I am thinking that it has to do with our implementation. Our implementation did not use a visitor pattern like some people did, but overall, I would say that our code was pretty efficient. Also, on average, the clang times were faster than our compiler code without any optimizations.

**Analysis**

For register use, ideally, we would only want to push/pop the registers that we are going to use. However, we made it so that the code just pushes and pops all of the r4-r10 registers for simplicity. This is one thing that simplifies the code, but it is costly in terms of performance.

The compiler is a little buggy for complex programs. For milestones 2 and 3, however, it works on all benchmarks. Past that, not all of the benchmarks pass or even compile correctly.

**Calling Convention**

Other than the fact that we pushed the r4-r10 registers onto the stack instead of choosing which ones are used, the caller saved registers are required to be saved and restored by the caller. The r4-10 registers, since they must be pushed onto the stack first and then popped off at the end of the function, require 14 memory operators per function call, which is not efficient at all.

**Conclusion**

In conclusion, compilers are hard to create and perfect. Making the compiler initially was not the terrible part. Optimizing the compiler and using registers instead of the stack are the hard parts. Our compiler, overall, works as a compiler is expected to work; however, the thing is that it is not fast at all. This class, in general, was very useful because knowing how something works on a lower level allows us to become better programmers as a whole even on the higher level end.