Cool Compiler Project

Ivan Rodriguez

Supervised by Dr. Rakesh Verma

1. Tips for Future Students  
     
    I would recommend being extremely comfortable with a high-level language like C++. I chose to do this project in C/C++, but I don’t think this is necessary. There are numerous ports of the flex and bison tools for languages like Python, OCaml, JavaScript, etc. That being said, a recruiter I spoke to was impressed that I did it in C/C++. If you do decide to go with C/C++, it is a MUST to learn GNU Make. I would go ahead and learn CMake as well, so that your project is more portable. I would also read in detail about the way C/C++ compiles programs. This means learning about linkers, loaders, static/dynamic libraries, etc. A great deal of time was spent on resolving build errors due to my inexperience with CMake. As mentioned above, I used flex and bison for the scanner and parser, respectively. These tools do not exactly have an easy-to-use interface. At one point, I tried using lemon (maintained as part of SQLite). I ended up switching back to bison, but lemon was also a great choice. Whatever tools you decide to use, get used to reading the documentation pretty extensively. GNU tools always come with great tutorials/documentation, and those tools are quite mature. As for the reading material, I recommend the classical book “Compilers: Principles, Techniques, & Tools” by Aho, Lam, Sethi, and Ullman. This is the definitive and timeless text on this subject. If you are pursuing a self-study, do some exercises at the end of every section. That being said, you don’t have to read every section, but it flows quite well, and I read most of the sections up through Chapter 8. You should not read this book with the expectation that your compiler will exactly mirror the chapter structure. That is far from the truth, and compilers vary a great amount. It is better to get a concrete understanding of the ideas on a chapter-by-chapter basis. Seeing as the book cover is a metaphor for conquering complexity, you should definitely recognize that this is a modular project. This is typically split up into a lexer, parser, some intermediate representation, then a code generation off of the intermediate representation. However, the compiler phases as presented in the book don’t always match the tools. For example, in my implementation, bison calls flex and receives tokens one by one. This was confusing at first, but implementation always differs from the theory. Once you are past the lexing and parsing, most of the code will be handwritten by you so I would suggest some whiteboarding/design sessions before implementing.
2. Code Overview  
     
    Flex is used to generate a scanner. The lex.ll file that flex takes as input is pretty trivial. We just specify tokens with regular expressions. On encountering a string literal, we change states and have special processing rules. Likewise for comments. The parse.yy file that bison takes as input is more complicated but not by much. We just specify the COOL grammar as presented in the reference manual but make changes on the fly because the specification is not in pure BNF (and it also uses some regular expression notation). The AST is constructed bottom up (first node to begin being constructed is the last node to finish being constructed). Since I used an LALR(1) parser, I did not have to worry about shift-reduce or reduce-reduce conflicts. Most semantic actions in the parser amount to calling new to create an AST node.   
    The driver program in main.cpp proceeds in a logical way. First, we check for the option --ast flag. Then, we call ParserDriver::parse() which is the driving call for the parser. By the end of this call, we have constructed the AST of the program. If the --ast flag was specified, we use the pretty print utility to print it to the console. After that, we build a hardcoded AST representing COOL’s basic classes that come with the language. This is done in the call to ParserDriver::buildInternalsAst(). Next, we call ParserDriver::buildEnvs(). This method traverses the top three floors of the AST (program, class, and method nodes) and builds symbol tables for classes and methods. Note that local symbol tables (for let and case expressions) are not built here. Next, we call ParserDriver::populateClassImplementationMaps(). This builds an inheritance graph, and then runs a depth first traversal on the inheritance graph. The class and implementation maps are data structures that map a class to its attributes and methods, respectively. This is done as part of visiting every node in the depth first traversal of the inheritance graph. Note that overriding methods replace the overridden method in its original index (this is how I achieve runtime polymorphism).   
    Next we call ParserDriver::decorateAST(). This is the first traversal of the fully constructed AST. We traverse the tree by recursively calling the pure virtual method decorate() declared in the \_node class and implemented by every descendant of this class. As part of the traversal, we keep track of pointers that tell us what scope we are currently in. The most important aspects of the first traversal are assigning a static type to each expression and doing a semantic check for the various rules of the COOL language. For let and case expressions, this traversal is when we build the local symbol tables. After checking for outstanding compile time errors, we move on to the code generation.  
    The code generation is the second and final pass of the AST. This pass takes care of generating the LLVM IR. First, we declare external C functions such as malloc, printf, etc. Then we generate the support class LLVMString, taken and adapted from [here](https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/README.html). We then declare all structures and functions. The structure convention/layout is to first have a pointer to that class’s virtual function table, followed by a pointer to a string that denotes the class name, followed by all attributes in order of appearance (with inherited attributes coming first). The virtual function table convention is to start with a pointer to the assembly constructor corresponding to the object’s runtime type, then all other methods (with inherited methods coming first). Once this is done, we move on to the bread and butter of the code generation: traversing each method of each class and consequently each expression. Each node that inherits from \_expr has implemented the pure virtual function codegen(). Finally, we generate the LLVM program’s entry point, which instantiates the implicit instance of Main and calls Main.main(). Runtime checks are generated as appropriate, and runtime errors include information about where the error occurs (line number).
3. Test Overview  
     
    Each part of the compiler is tested independently using the reference compiler. The tests are actually just simple diffs because each phase has a serialization specification that can be found [here](https://dijkstra.eecs.umich.edu/eecs483/cool.php), under Assignments. The framework I used is Google Test, which is just a C/C++ implementation of the [xUnit](https://en.wikipedia.org/wiki/XUnit) architecture. The test executable is a separate module and is run independently of the main module. The tests for the lexer and parser are trivial, since those only test for the structure of a program. The tests for the semantic analyzer and code generation are of the most importance. For the semantic analyzer and code generation, I’ve included both positive and negative test cases, and each semantic error or runtime error has at least one corresponding test case. The test project automates running the generated LLVM IR file through clang and then running the executable generated from that LLVM IR file.
4. Links  
     
   A read only link to my notes and exercises from the text: <https://1drv.ms/u/s!AmpI6M1P5deIg2ABkLA41quxlK3G>  
   A link to the University of Michigan Compiler Construction course website: <https://dijkstra.eecs.umich.edu/eecs483/index.php>  
   A link to a recording of the live demo from May 10th:  
   <https://drive.google.com/file/d/1xYYL4y4zftFCEKQa4g9tsvxff3d4cdm5/view?usp=sharing>  
   A link to the GitHub repo of the project’s source code:   
   <https://github.com/ivanrodriguez3753/CoolCompilerProject>

A link to the merged pull request I made to mapping-high-level-constructs-to-llvm-ir:  
<https://github.com/f0rki/mapping-high-level-constructs-to-llvm-ir/commit/659052d3a055ff9cdf81223e4842ad13aab6bc59>