Esther Fatehi, fatehie@oregonstate.edu
Josh Huff, huffj@oregonstate.edu
Jose Murrieta, murrietj@oregonstate.edu
Caelum - CMP1
Instructor Brewster
CS 467 – Spring 2018
June 8, 2018

Capstone Project Final Report: COOL Compiler

Introduction

Ten weeks ago, Caelum accepted the task of developing a compiler as described by Stanford University's online course "Compilers." Today, we are proud to announce that our development process has reached completion. This document discusses its function from the user's perspective, provides step-by-step instructions with helpful screenshots to help a user get started, describes how the software system's internal modules coordinate, lists the technologies and knowledge required during the course of development, and enumerates some (but certainly not all!) of the team members' individual contributions and challenges.

What It Does

In short, Caelum's compiler transforms a COOL source file into its equivalent in MIPS assembly language. The target user for this application has at least a passing familiarity with the concepts of statically-typed, compiled languages and with the use of a command line interface. The compiler command, "mycoolc", accepts one argument: the pathname of a file written in a language called COOL (Classroom Object Oriented Language) and creates a target file in the same directory. This new file has the same name as the source file, except for its file extension, ".s", which indicates it contains MIPS assembly code. Of course, the above assumes the COOL file exists and is syntactically and semantically correct. In the event that a COOL file misuses keywords, violates the COOL inheritance hierarchy, or otherwise fails to conform to the language standard, the compiler will cease and produce a corresponding error message instead of the expected target file. If that is the case, then a user must correct the source file's deficiencies and re-attempt compilation.

An important fact about the COOL compiler: because type safety is paramount, the compiler's typing system values soundness over completeness. Soundness is the principle that the compiler will never allow an invalid source file to achieve compilation. Completeness is the principle that the compiler will never reject a source file that would correctly compile. In practice,

the compiler rejects programs with phrasings that fail to conform to type-checking rules even if the actual code would evaluate to a correct type. This preference exists to absolutely preclude the possibility of undefined behavior.

How It Works

Typically, production compilers have optimization phases and sundry features that enhance the user experience and aid in productivity. COOL was designed to facilitate understanding of the theoretical principles of compilation and its implementation. As such, the COOL compiler operates in the canonical, sequential phases. Each phase is controlled by its own module. These are the lexer, the parser, the semantic analyzer, and the code generator.

The lexer uses regular expressions to identify substrings of the source file's text that constitute lexical units (called "lexemes"). To build our lexer, we utilized a tool called flex that allows you to write rules that match on defined regular expressions and perform a specified action for each matched pattern. Example lexemes include type identifiers, object identifiers, keywords, and so on. These are packaged into discrete units and passed on to the parser. The colloquial description of this process is "tokenization." At this stage, a file with an invalid character (one that can’t begin any token as specified by the language) or an illegal string constant (one that is too long, is unterminated, or contains the null character) will cause compilation to fail.

The parser takes the lexemes and leverages the GNU tool Bison to construct an abstract syntax tree (AST). The AST structure is absolutely crucial for the proper function of the remaining phases of compilation. Therefore, if the tree is malformed, it is the parser that identifies the discrepancy and halts execution. Otherwise, the completed AST is passed to the semantic analyzer.

The semantic analyzer takes the AST and recursively traverses it in multiple passes, first to verify that each node representing a class helps constitute a valid hierarchy, then to "decorate" it by setting each node's type to the value determined to be appropriate as a result of COOL's type system. If there is an error in hierarchy, scope, or naming, execution halts. Otherwise, the type-decorated AST is passed to the final phase.

The code generator takes the AST and recursively traverses it, using the data from each node to produce assembly language instructions. At this point, there should be no errors in syntax or semantics, because spim, the MIPS simulator, has no concept of type and simply performs the low-level operations on each node as its instructed. The finished product is a file containing the MIPS equivalent of the COOL file's content.

Assuming all these phases successfully complete, the resultant file can be executed whenever the user desires.

Getting Started With Caelum’s COOL Compiler

Managing a complex set of dependencies is difficult and tedious enough for one given platform, but the nature of the COOL compiler is such that attempting to develop and run it on even slightly different environments would be a recipe for frustration. Our team has elected to eliminate that possibility and achieve total platform independence.

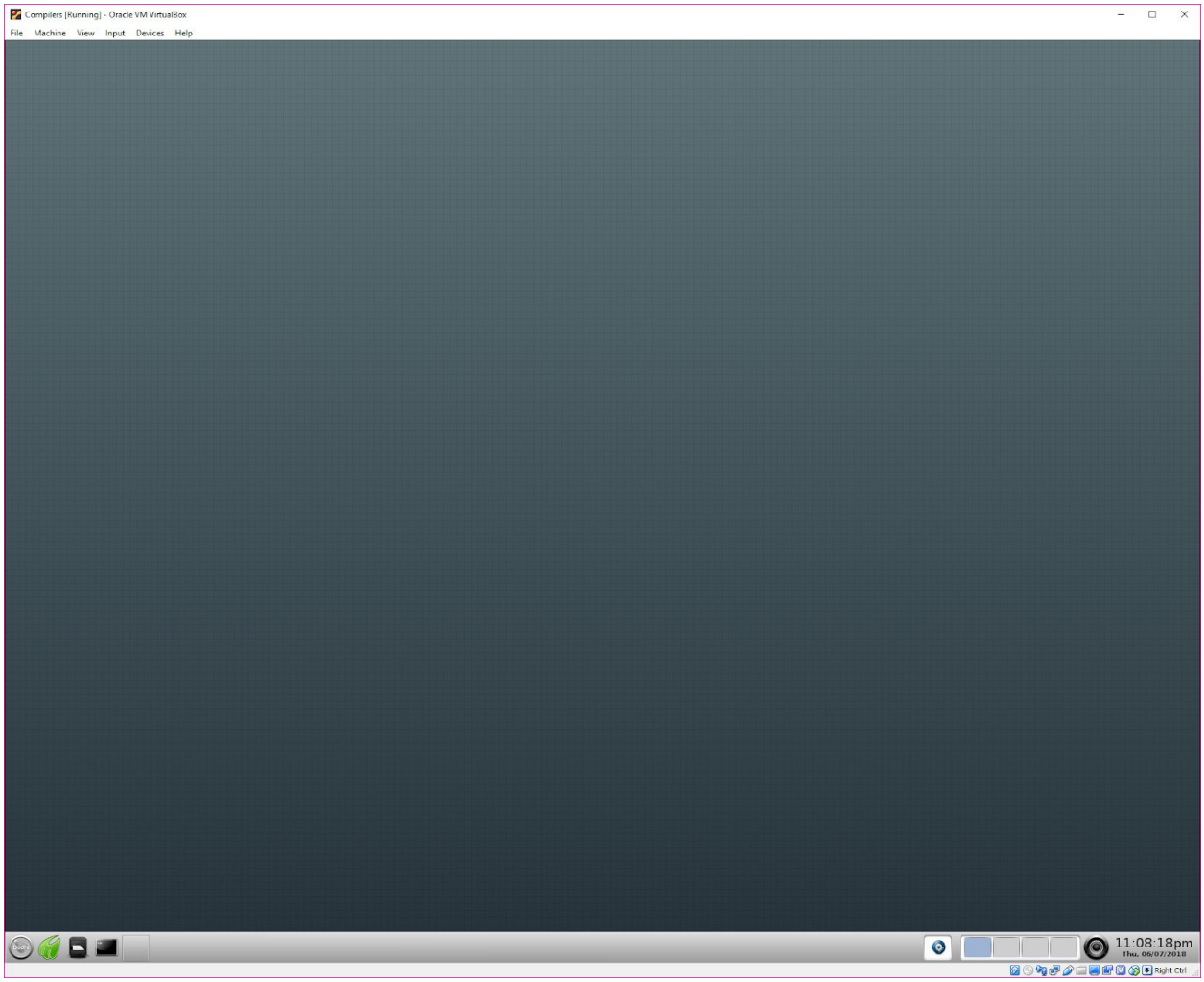
Consequently, instead of being conceived as a standard executable, Caelum's COOL compiler has been developed and hosted entirely on a Linux virtual machine (VM). The exact distribution is a stripped-down offshoot of Ubuntu called "Bodhi." This design choice ensures that the application can be run on literally any machine that can support a VM. However, the tradeoff for this convenience is heavy resource consumption. According to the Stanford Compiler course author, the VM takes a bit more than 512 MB of RAM and 2 GB of hard disk space. Furthermore, it is recommended that the host machine have closer to 2 GB total RAM to comfortably accommodate it.

If your workstation does not already have Oracle's VirtualBox, you must first download and install it. Installing and operating VirtualBox is relatively simple, and there are a number of excellent resources for getting it up and running. Its official website has lots of documentation and free download options: https://www.virtualbox.org/

That's the only requirement; the rest of COOL's needs are on-board.

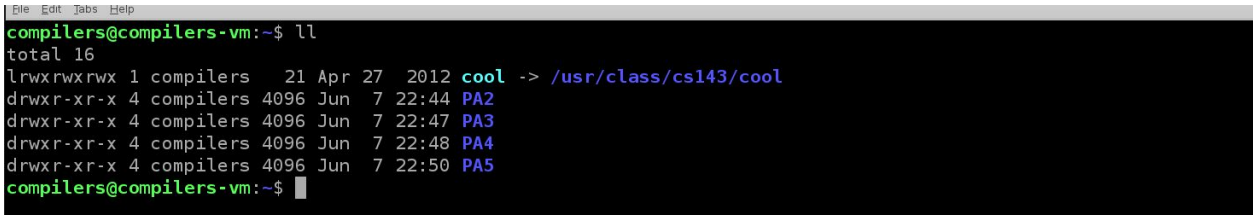
Step-by-Step Instructions

When you click the green Start arrow in VirtualBox to activate the VM, you will see a boot-up screen with several options. This does not require any thought or user input. Just hit Enter or simply allow the timer to elapse and the default (and correct) option will be chosen for you. When the VM is finished booting, you will be greeted with this screen.



The next and remaining steps in exploring the COOL compiler are all taken using LXTerminal, a terminal emulator. Its icon is the rightmost black square in the bottom left corner of the screen. Alternatively, you can reach the terminal at any time (and access Bodhi's other applications and system settings, if necessary) by left-clicking the desktop and directing the cursor from the pop-up menu's options "Applications" to "Accessories" to "LXTerminal."

Once in the terminal, enter the long list command by typing "ll" -- that's two lowercase Ls -- at the prompt. The result should look like this:



What you're seeing is the home directory of the Caelum compiler. To aid navigation and use, the hierarchy has been highly simplified from the course VM's original configuration.

The present directories are:

PA2 -- the lexer
PA3 -- the parser
PA4 -- the semantic analyzer
PA5 -- the code generator

To see the contents of each directory, you can use the long list ("ll") command followed by the directory name, like this: ll PA2

Alternatively, you can use the change directory ("cd") command to navigate to a specific folder like this: cd PA2

In either case, entering "cd" by itself to the terminal will return you to this home directory.

For the sake of demonstration, the components in the early phases of compilation -- the lexer and parser -- have been coupled with fully-functional reference modules. Taking PA2, the lexer, as an example, we can change into the directory and give the long list command to see that the lexer and the parser commands have both symbolically linked with the provided semantic analyzer and code generator. From the parser on, later components have been integrated with the team-created versions.

```
compilers@compilers-vm:~/PA2$ ll
total 684
lrwxrwxrwx 1 compilers     47 May  7 18:26 cgen -> /usr/class/cs143/cool/etc/../lib/.i686/PA2/cgen
-rw-r--r-- 1 compilers   6690 May 22 23:45 cool.flex
-rw-r--r-- 1 compilers  69786 May 22 23:45 cool-lex.cc
-rw-r--r-- 1 compilers    414 May 22 23:45 cool-lex.d
-rw-r--r-- 1 compilers  78432 May 22 23:45 cool-lex.o
drwxr-xr-x 2 compilers   4096 Jun  7 22:41 examples
drwxr-xr-x 3 compilers   4096 Jun  7 22:22 grading
lrwxrwxrwx 1 compilers     45 May  7 18:26 handle_flags.cc -> /usr/class/cs143/cool/src/PA2/handle_flags.cc
-rw-r--r-- 1 compilers    240 May 22 23:45 handle_flags.d
-rw-r--r-- 1 compilers  43212 May 14 14:16 handle_flags.o
-rwxr-xr-x 1 compilers 184585 May 28 22:53 lexer
lrwxrwxrwx 1 compilers     40 May  7 18:26 lextest.cc -> /usr/class/cs143/cool/src/PA2/lextest.cc
-rw-r--r-- 1 compilers    411 May 22 23:45 lextest.d
-rw-r--r-- 1 compilers  43968 May 14 14:16 lextest.o
lrwxrwxrwx 1 compilers     53 May  7 18:26 Makefile -> /usr/class/cs143/cool/etc/../assignments/PA2/Makefile
lrwxrwxrwx 1 compilers     37 May  7 18:26 mycoolc -> /usr/class/cs143/cool/src/PA2/mycoolc
-rwxr-xr-x 1 compilers  62171 Aug 29  2014 pa1-grading.pl
lrwxrwxrwx 1 compilers     49 May  7 18:26 parser -> /usr/class/cs143/cool/etc/../lib/.i686/PA2/parser
-rw-r--r-- 1 compilers   3270 May  7 18:26 README
lrwxrwxrwx 1 compilers     49 May  7 18:26 semant -> /usr/class/cs143/cool/etc/../lib/.i686/PA2/semant
lrwxrwxrwx 1 compilers     42 May  7 18:26 stringtab.cc -> /usr/class/cs143/cool/src/PA2/stringtab.cc
-rw-r--r-- 1 compilers    382 May 22 23:45 stringtab.d
-rw-r--r-- 1 compilers  90304 May 14 14:16 stringtab.o
-rw-r--r-- 1 compilers   2488 May  7 18:26 test.cl
lrwxrwxrwx 1 compilers     42 May  7 18:26 utilities.cc -> /usr/class/cs143/cool/src/PA2/utilities.cc
-rw-r--r-- 1 compilers    513 May 22 23:45 utilities.d
-rw-r--r-- 1 compilers  59384 May 14 14:16 utilities.o
compilers@compilers-vm:~/PA2$
```

There's no need to learn the COOL language to see the lexer in action. For the sake of convenience, each component's directory has its own set of COOL language example files. To work through an example, run either (“ll”) or the standard list command ("ls") on the given module's examples subdirectory.

```
compilers@compilers-vm:~/PA2/examples$ ll
total 104
-rw-r--r-- 1 compilers 10222 Jun  7 22:41 arith.cl
-rw-r--r-- 1 compilers  2738 Jun  7 22:41 atoi.cl
-rw-r--r-- 1 compilers  1073 Jun  7 22:41 atoi_test.cl
-rw-r--r-- 1 compilers  3510 Jun  7 22:41 book_list.cl
-rw-r--r-- 1 compilers  2467 Jun  7 22:41 cells.cl
-rw-r--r-- 1 compilers   707 Jun  7 22:41 complex.cl
-rw-r--r-- 1 compilers   200 Jun  7 22:41 cool.cl
```

Then choose a COOL source file such as hello_world.cl and use that file's name as an argument when invoking the "mycoolc" command.

```
compilers@compilers-vm:~/PA2$ ls examples/
arith.cl  atoi_test.cl  cells.cl    cool.cl   hairyscary.cl   io.cl   life.cl  new_complex.cl  primes.cl  sort_list.cl
atoi.cl   book_list.cl  complex.cl  graph.cl  hello_world.cl  lam.cl  list.cl  palindrome.cl   README
compilers@compilers-vm:~/PA2$ mycoolc examples/hello_world.cl
compilers@compilers-vm:~/PA2$ ls examples/
arith.cl  atoi_test.cl  cells.cl    cool.cl   hairyscary.cl   hello_world.s  lam.cl   list.cl        palindrome.cl  README
atoi.cl   book_list.cl  complex.cl  graph.cl  hello_world.cl  io.cl          life.cl  new_complex.cl  primes.cl      sort_list.cl
compilers@compilers-vm:~/PA2$
```

Notice that, after the mycoolc command has been entered, the examples subdirectory is one file heavier. This new entry, hello_world.s, is in our target language of MIPS assembly, and is obviously the result of our compilation. Now all that's left to do is execute the file using spim, a program that simulates the functionality of a MIPS processor. Invoke the "processor" by simply giving the command "spim" followed by the location of the MIPS assembly file, like this: spim examples/hello_world.s

```
compilers@compilers-vm:~/PA2$ ls examples/
arith.cl  atoi_test.cl  cells.cl    cool.cl   hairyscary.cl   i
atoi.cl   book_list.cl  complex.cl  graph.cl  hello_world.cl  l
compilers@compilers-vm:~/PA2$ mycoolc examples/hello_world.cl
compilers@compilers-vm:~/PA2$ ls examples/
arith.cl  atoi_test.cl  cells.cl    cool.cl   hairyscary.cl   h
atoi.cl   book_list.cl  complex.cl  graph.cl  hello_world.cl  i
compilers@compilers-vm:~/PA2$ spim examples/hello_world.s
SPIM Version 6.5 of January 4, 2003
Copyright 1990-2003 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/class/cs143/cool/lib/trap.handler
Hello, World.
COOL program successfully executed
Stats -- #instructions : 152
         #reads : 27  #writes 22  #branches 28  #other 75
compilers@compilers-vm:~/PA2$
```

As you can see, buried in between the copyright information for spim and its execution statistics for the given sample is the expected output: Hello, World.

To verify that this is, indeed, the expected output, we can run the provided reference compiler under the same conditions and observe the exact same result. The only difference is the compiler invocation: instead of "mycoolc", use "coolc"

The process described above can be applied to all other individual components.

If you care to explore the compiler's ability to catch syntax errors, you could try "commenting out" an important line using two dashes "--" in the same way C-family languages use two slashes in **cool.cl**, by navigating to line 10 and inserting a comment to obscure the semi-colon or change out_string to "ot_string" and see that attempted compilation gives a syntax error.

If you care to explore the compiler's ability to catch semantic errors, try violating an inheritance rule in **io.cl**, by changing line 60 so that class B inherits B. Compilation will halt, informing you that there is an inheritance cycle.

If you care to explore the compiler's ability to generate something more than "Hello, World" text, try **primes.cl:** in it, there is a "halt" string provided to stop execution at 500, and accordingly, the primes.s file calculates and outputs all prime numbers less than 500.

Required Knowledge and Technologies

Instead of using APIs or frameworks, our support comes in the form of skeleton files provided as part of the Stanford course content. At first glance, these files seems structurally complete. Such was certainly not the case; the latter phases required polymorphic functions and helpers to effectively access AST node content. The modules are based in, and interact via, C++. The first phases required assistance from tools like Bison and flex, and the code generation phase is largely written in C++ which generates MIPS assembly code.

Unfortunately, while the documentation regarding the compiler course VM indicated that it would be possible to download additional packages to aid in development, none of the team members could successfully install version control software like Git or debugging tools like GDB. To overcome the lack of Git, team members were forced to revert to pre-VCS means of tracking changes (e.g. creating multiple copies of a file with names describing their respective states, or copy-pasting files between host and VM so the host could interface with GitHub. A similar solution for GDB wasn't possible without introducing the potential for a platform difference issue, and without access to GDB, team members were left with only rudimentary debugging techniques like trace statements, which often proved inadequate for recursive and stack-based operations like the compiler heavily depends on.

Caelum Team Member Accomplishment

All Members

The COOL compiler is an unusual Capstone project in that nearly half of the time available for development was spent watching lectures, reading domain-specific documentation, and locating additional resources to get up to speed with the theory of compilation and becoming functionally knowledgeable about COOL. The Caelum team divided the entire ten week lectures series into two week-long cram sessions. It bears mentioning that the “weeks” of lectures were no less than 90 minutes each. It's arguable that our team of three operating and delivering what is expected of a single undergraduate with the same development schedule is a textbook observation of Brooks's Law.

Simply put, the overhead in getting started was staggering. While there were provided tools that certainly made project completion within the deadline achievable, the tools themselves required additional study and practice to learn how their place in implementation. For each of us, this was a first non-trivial exposure to metaprogramming, and it was difficult to overcome the semantic satiation of discerning, for example, Class_ from CLASS from class.

Jose Murrieta: Lexer and Parser

Jose implemented the initial two phases. Both the lexer and parser achieve a 100% success rate when tested using their respective grading scripts, and they both consistently approve (or, as appropriate, reject) all of the example files provided in the course VM. In addition to the required knowledge common to all members, Jose's duties required him to become proficient in flex, Bison, and regular expressions. Jose also prepared the demo VM, contributing to its user-friendliness.

Josh Huff: Semantic Analyzer

Huff implemented the third phase. The semantic analyzer achieves a 98% success rate when tested with its grading script. In addition to the required knowledge common to all members, Huff's duties required an understanding of the symbol table data structure, and the principles of conformance and the logical rules of inference in the context of the COOL language specification.

Esther Fatehi: Code Generation

Esther implemented the fourth and most difficult phase. The code generator can successfully transform hello_world.cl, cool.cl, io.cl, primes.cl, complex.cl, and new_complex.cl. In addition to the required knowledge common to all members, Esther's duties required a working understanding of the runtime system and MIPS assembly, itself substantially different from the assembly dialect with which she is familiar.

Conclusion

Leonardo Da Vinci is credited with the phrase, "Art is never finished, only abandoned." By no stretch of the imagination do we fancy our project a work of art, but time compels us to abandon this labor before we'd prefer. As is the case for all non-trivial software, Caelum's COOL compiler has faults. Despite the challenges described above and the myriad technical issues we encountered, each of us managed to produce a working module that coheres with the others. The client has specified the need for the project to require at least 100 hours of effort on the part of each member over the 10-week development period. It is not an exaggeration, but a conservative estimate, that each team member has contributed that much time in service of the group effort in the last three weeks alone.