Loyola Marymount University
Frank R. Seaver College of Science and Engineering

Ekko

Harnessing the Power of Javascript for Rapid and Accessible Compiler Development

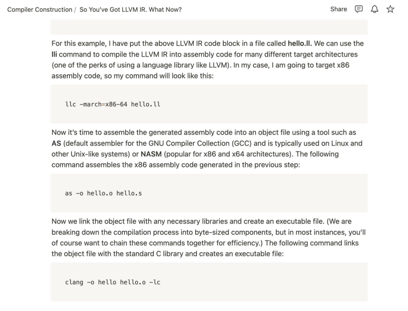
Authored by Kira Toal
Computer Science Department
Spring 2023

Motivation

LLVM is a popular and versatile compiler infrastructure that supports multiple programming languages. Languages with compilers that target LLVM include C#, CUDA, Julia, Kotlin, Lua, Objective-C, Ruby, Rust, and Swift. Of the languages that target LLVM, many also utilize Clang, which provides a compiler front-end and tooling infrastructure for programming languages. Although Ekko targets LLVM, its front-end is implemented entirely in JavaScript. This project serves as an exploration into the utility of using non-C-based, high-level programming languages for front-end compiler development. Despite some performance limitations, JavaScript's rapid prototyping capabilities, interoperability, and mature ecosystem of development tools (including the Ohm parsing toolkit) proved that it merits further investigation in this space.

Implementation

Ekko is a statically typed, compiled language that provides the standard set of features commonly found in programming languages, encompassing variables, operators, control flow structures, and functions, among others. Its syntax is defined using the Ohm language library. Once an Ekko program is parsed into an AST, a custom-built analyzer identified semantic errors and performs optimizations. The Ekko generator includes modules for each grammar rule that generate corresponding LLVM IR — a low-level, platform-independent representation of source code that serves as a bridge between the JavaScript front-end and the LLVM back-end. After receiving LLVM IR, the LLVM compiler backend performs its own optimizations and generates target-specific machine code, which can be executed directly on a target architecture or further processed.



Compiler Construction / So You've Got LLVM IR. What Now?   Share

For this example, I have put the above LLVM IR code block in a file called **hello.ll**. We can use the **lli** command to compile the LLVM IR into assembly code for many different target architectures (one of the perks of using a language library like LLVM). In my case, I am going to target x86 assembly code, so my command will look like this:

```
llc -march=x86-64 hello.ll
```

Now it's time to assemble the generated assembly code into an object file using a tool such as **AS** (default assembler for the GNU Compiler Collection (GCC) and is typically used on Linux and other Unix-like systems) or **NASM** (popular for x86 and x64 architectures). The following command assembles the x86 assembly code generated in the previous step:

```
as -o hello.o hello.s
```

Now we link the object file with any necessary libraries and create an executable file. (We are breaking down the compilation process into byte-sized components, but in most instances, you'll of course want to chain these commands together for efficiency.) The following command links the object file with the standard C library and creates an executable file:

```
clang -o hello hello.o -lc
```

Documentation

While compilers are an essential component of modern software development, compiler development is considered a relatively niche field within software engineering due to its specialized nature. Similarly, while LLVM is a powerful tool, it may have a high barrier of entry for some developers, depending on their background and experience. To lower this barrier, the author has documented the development of Ekko on a Notion blog so that engineers without significant experience might avoid common pitfalls. The Notion can be found on the project's Github page.

Future Development

The Ekko programming language was named in part after a time-traveling character from Netflix's Arcane/Riot Games' League of Legends and in part after a another "E" language, Elm, which supports time travel debugging. Ekko was originally intended to feature a time-travel debugging state which the user could enter directly from the console. The focus of the project shifted to synergizing Ohm, JavaScript, and LLVM, but future development will center the integration of temporal debugging features such as value histories and time-aware watchpoints.

A Simple Example

One of the simplest syntactically correct programs that can be written in Ekko the following:

```
examples > print.ekko
print("Hello, Zaun!")
```

The Ekko program is first parsed into an AST according to the rules defined in its grammar. After semantic analysis and optimization, the AST for this small program is as follows:

```
kira@Kiras-MacBook-Air ekko % node src/ekko.js
 examples/print.ekko analyzed
   1 | Program statements=[#2]
   2 | PrintStatement value='Hello, Zaun!'
```

The optimized AST is then passed to the generator, which produces LLVM IR. The `PrintStatement(p)` function in the generator, for example, pushes to the `output.ll` file the LLVM IR required to print a user-specified string to the console.

```
PrintStatement(p) {
  const nullTerminatedString = `"${p.value}\\00"`; // Append null terminator
  const length = p.value.length + 1;
  const llvmIR = `
    @.str = private unnamed_addr constant [${length} x i8] c${nullTerminatedString}

    declare i32 @puts(i8*)

    define i32 @main() {
      %1 = getelementptr [${length} x i8], [${length} x i8]* @.str, i32 0, i32 0
      call i32 @puts(i8* %1)
      ret i32 0
    }
  `;
```

`output.ll` can be easily integrated into the LLVM pipeline. The following bash script is run to chain together the front and back-end portions of the compilation process:

```
node src/ekko.js examples/$1.ekko js
# Define path to ll file.
path_to_ll="$(pwd)/output.ll"
llc -march=x86-64 $path_to_ll
# Assemble the generated assembly code into an object file.
as -o output.o output.s
# Link the object file with any necessary libraries and create an executable file.
clang -o output output.o -lc
./"output"
```

```
kira@Kiras-MacBook-Air ekko % ekko print
Hello, Zaun!
```