Capstone Project Report: Developing a Custom Programming Language Using LLVM

Angel Daniel Colon

SUNY Polytechnic Institute

CS 498: Capstone Project

Advisor: Dr. “Amos” Confer

October 12, 2024

Capstone Project Report: Developing a Custom Programming Language Using LLVM

The purpose of my capstone, my final major project, was to write a fully functional, custom programming language using the Low-Level Virtual Machine (LLVM). The language had to handle arithmetic, recursion, and user-defined functions and allow the user to define logical operators of their choice. The development process involved writing a lexer, parser, and code generator that converted the source code into LLVM Intermediate Representation (IR) before calling on utility functions to compile IR into machine code. This report details the language's architecture, my challenges during implementation, and what I learned from them.

The core of this project was building a functional programming language that could utilize LLVM to generate machine code—designed to prioritize immutability and make functions first-class citizens. Functional programming is suitable for both mathematical computations and recursion-based programming. This project was possible because LLVM is a compiler framework: it is highly optimized to generate efficient but universal machine code for all types of computers.

LLVM takes high-level source code and compiles it into LLVM IR, a simple, intermediate representation (IR) that can be optimized and compiled into machine code. So, with LLVM’s modular design, I developed the front end of the language that reads code, and LLVM can handle the complexities of code generation and optimization for me.

This process had several iterations. The first step was the lexical analysis, separating the source code into tokens, basically, the minor linguistic units that make up a language. The tokens

were subsequently parsed to create an abstract syntax tree (AST), which provided the language in which the program was written in its hierarchical structure. Finally, the AST was translated into LLVM IR and machine code.

This language allows most standard functional programming features, such as user-defined functions, recursion, and arbitrary loops. You can write mathematical functions, define new algorithms, and use recursion. A typical function could be something like a Fibonacci sequence:

```
def fib(x)
  if x < 3 then
    1
  else
    fib(x-1) + fib(x-2);
fib(4)
```

Such a minimal and evocative syntax gives users a simple but powerful pattern for describing mathematical functions and recursive algorithms.

The language is designed around these three pieces: a lexer, a parser, and a code generator. The first stage of a compiler is called the lexer, and it takes the source code and breaks it up into a stream of tokens. This is how a computer stops reading the letters and symbols in a text file and starts recognizing functional components of code that are necessary to process it for a specific language. A token can be a function definition, an operator, a literal, or something similar. For instance, the function def fib(x) will be broken up into tokens of the keywords def and the identifier fib.

The parser accepts the tokens created by the lexer and establishes a meaning hierarchy represented in an abstract syntax tree (AST). The hierarchical relationships between program entities, such as, for example, the parameters of a function, the operators that work on those parameters, and the expressions that combine the operators' results, are represented by the AST as the path from the tree's leaves to the root. Consider, for example, the function that calls itself recursively. If we represent it in AST, the hierarchical logic of the function's call and return result from the conditions will be mapped to the tree structure. The code generator then converts the AST into LLVM IR, a low-level representation abstract enough to represent much higher-level constructs as an intermediate representation. From there, LLVM IR is optimized and compiled into machine code capable of executing on the hardware's instruction set. For instance, the code generator employs libraries of interfaces (called 'APIs') provided by LLVM to translate each node in the AST into the appropriate LLVM instruction that will execute the program correctly and efficiently.

Adding logical operators was another of the project's most challenging tasks. I first tried to add logical AND (&), OR (|), and NOT (!) in the language. However, assigning tokens to these operators in the lexer, parsing them correctly, and ensuring they were integrated into the code generation process proved difficult. Specifically, managing the precedence of logical operators alongside other operators created significant complexity in both the parsing and code generation stages.

After several attempts to solve this issue, I left the implementation of logical operators to the users. In other words, basic arithmetic operators are built into the language, while logical operators are not. A user can define them on their own. For example, to specify a logical AND operator, one might say:

```
def binary& (LHS RHS)
  if !LHS then
    0
  else
    !!RHS;
```

This approach meant I could keep the language implementation simple without cutting off the extension of the features of the language to the users as much as I could. Beyond the issues with writing the quantifiers, which I painstakingly explained over a long-winded three-part series (if that's any indication, these weren't easy problems!), I encountered challenges in getting LLVM configured and honed to be able to integrate it into my language's code generation stage. It took me some time to debug even the configuration and linking issues, but eventually, I was able to solve them and check an oversized item off my to-do list that needed to be there for the language to be able to compile any code.

Then, there was the memory management issue, where recursive function calls again tackled me. If not correctly handled, recursion can lead to a burst of stack growth, and I wanted to ensure that the LLVM-generated code could handle it efficiently. I leveraged LLVM's optimization capabilities to compile recursive functions into optimized machine code that avoided stack blooms.

If I embark on this type of project again, I might adopt an incremental approach to implementing core features. One of the things I could have done better early on was to try to implement both the arithmetic and logical operators at once, which made debugging and general testing a nightmare. Ultimately, implementing the arithmetic operators individually and testing them one by one before adding the yet more complex constructs that the logical operators give rise to would have made things much easier and less error-prone.

Lastly, I would focus on improving its error-handling functionality. Currently, the language has default diagnostic messages for everything from syntax to runtime errors, but it would be helpful to supply further verbose messages that aid the user in resolving issues with their code. For example, diagnostic messages that provide more descriptive information would be beneficial in cases where a function is misused or when a native operator that the user-defined is undefined.

My final year project was a fantastic opportunity to develop a small, custom functional programming language using LLVM, a built-on top-of-the-art, low-level intermediate representation infrastructure. Through this project, I applied the knowledge and skills learned in my computer science degree to a real-world problem. Additionally, my project fuelled my interest in learning more about compiler design and functional programming, and it provided me with hands-on experience with LLVM.

My efforts, especially concerning logical operators and the LLVM configuration, forced me to think creatively and sharpen my problem-solving skills. By the end of the project, I had

developed a working programming language that could parse and execute user-defined functions, have a mechanism to deal with recursion, and even allow users to extend their functionalities with their logical operators. The project has helped me grow as a developer and sharpened my understanding of programming language design and compiler builds.

Taken as a whole, the project is a good combination that completes my degree in Computer Science because it gives an overview of my ability to develop complex software systems. It also emphasizes applying the theory learned in my studies, demonstrating a practical and experimental application of what my professional career will demand: to tackle complex issues under challenging contexts in the technological domain. The skills and understanding from this project will benefit my future projects.

References

LLVM Project. (n.d.). *Kaleidoscope: Implementing a language with LLVM* [Tutorial]. LLVM. https://llvm.org/docs/tutorial/index.html