Prof. Monica Shah
Compiler Construction

How To Make A Programming Language

With LLVM and ANTLR4

Aayush Shah 19BCE245
Sachi Chaudhary 19BCE230

Index

INTRODUCTION

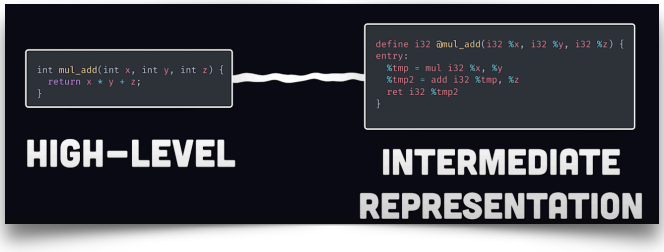
Every programmer might have thought about building own programming language at some point in their life. We wanted to do the same.

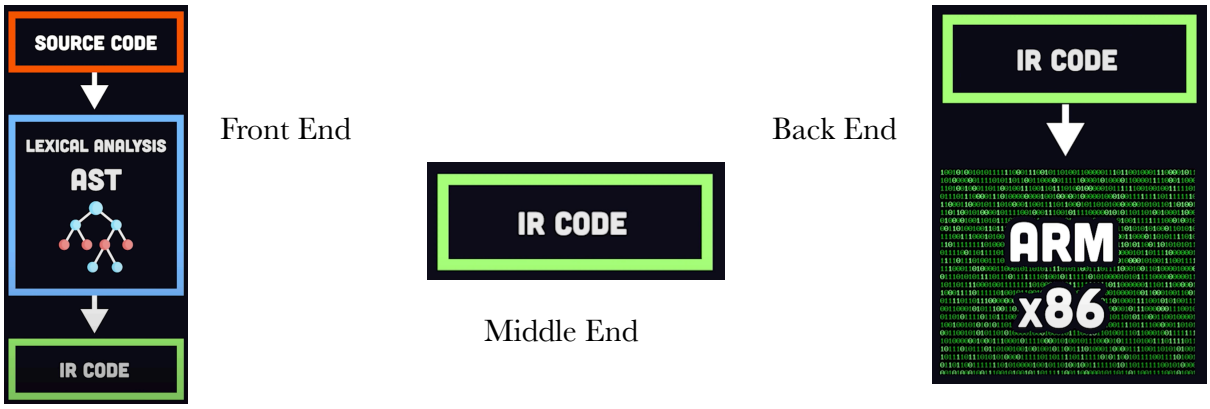
As it seems, It will not be a cup of cake but we'll do it for sure. Also it depends on how complex you want your language be. With this guide, hopefully you'll have a better concept of how to implement your language. At the time of writing, We'll be employing newer technologies such as ANTLR4 and LLVM, which thankfully do a lot of the grunt work for us. So, with everything out the way, let's get started!

What is LLVM ?

LLVM a toolkit used to build and optimise compilers. Building a programming language from scratch is hard. You as humans who want to write code in a nice simple syntax than machines that need to run it on all sorts of architectures, LLVM standardises the extremely complex process of turning source code into machine code. It was created in 2003 by grad student Chris Lattner at the University of Illinois and today it's the magic behind clang for C and C++ as well as languages like Rust, Swift, Julia and many others.

Most importantly it represents high level source code in a language agnostic code called intermediate representation or IR. This means vastly different languages like Cuda and Ruby produce the same IR allowing them to share tools for analysis and optimisation before they're converted to machine code for a specific chip architecture.



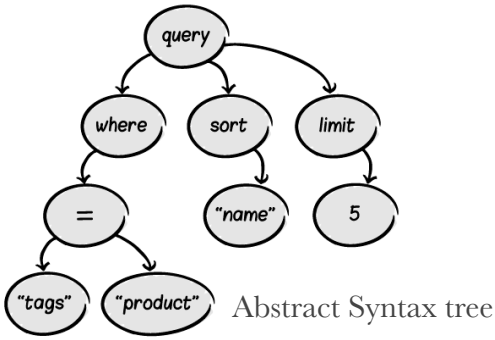


A compiler can be broken down into three parts: The front end parses the source code text and converts it into IR. The middle end analyses and optimises this generated code and finally the backend converts the IR into native machine code.

To build your own programming language from scratch right now :

1. Install LLVM then create a .c file now. Envision the programming language syntax of your dreams to make that high level code work.
2. **LEXER** *(turn raw text into tokens)* : You'll first need to write a Lexer to scan the raw source code and break it into a collection of tokens like literals identifiers keywords operators and so on next we'll need to define an Abstract Syntax Tree to represent the actual structure of the code and how different tokens relate to each other which is accomplished by giving each node its own class.

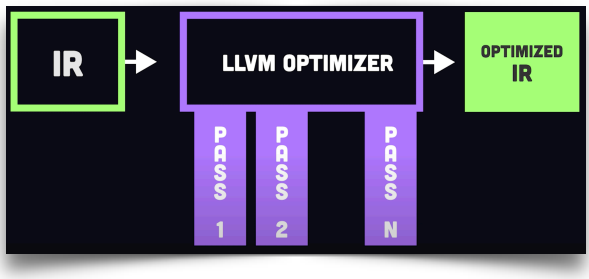
Token name	Sample token values
identifier	x, color, UP
keyword	if, while, return
separator	}, (, ;
operator	+, <, =
literal	true, 6.02e23, "music"



Abstract Syntax tree

3. **Parser** *(construct the AST)* : Third, we need a parser to loop over each token and build out the abstract syntax tree if you made it this far, congratulations! because the hard part is over.
4. **IR** *(generate intermediate representation code)* : Now we can import a bunch of LLVM primitives to generate the intermediate representation each type in the abstract syntax tree is given a method called codegen which always returns an LLVM value object used to represent a single assignment register which is a variable for the compiler that can only be assigned once. what's interesting about these IR primitives is that, unlike assembly they're independent of any particular machine architecture and that dramatically simplifies

things for language developers who no longer need to match the output to a processor's instruction set. Now that the front end can generate IR the op tool is used to analyse and optimise the generated code it makes multiple passes over the IR and does things like *dead code elimination* and *scalar replacement of aggregates* and finally that brings us to the back end where we write a module that takes IR as an input that emits object code that can run on any architecture. Congratulations, you just built your own custom programming language and compiler. Anyways, let's dive deeper!



References :

- LLVM official guide for language frontend building