Towards More Streamlined Compiler Construction

Jagger De Leo

April 2019

1 Abstract

Compiler construction has a semantics problem. Tools already exist to address many of the common issues that are encountered in the process of creating a new language. This paper examines popular compiler construction tools and makes some suggestions as to where improvements can be made. With a little effort, one can define a concise and complete language grammar. Likewise, powerful tools like LLVM exist to create highly optimized, multi-platform machine code through use of an intermediate code representation. There is, however, a lack of tools available for use in defining language semantics. Significant manual work must be done in a helper language, such as C++, to give semantic meaning to a program's parse tree. A case study of the implementation of a programming language was done to identify shortcomings in the current range of tools.

2 Introduction

The job of building a compiler is multifaceted–arguably multidisciplinary. A single programmer has to understand a wide array of ideas. At a high level, a programmer needs to have at least a basic understanding of formal grammars, text parsing, logical structures, and machine code generation for the plethora of different targets and platforms. Creating a user friendly programming language that performs well is a significant undertaking.

It used to be that programmers could write their own compilers almost as a rite of passage. Today things are harder. Over the years, computers have become more complex, and so has their software. Compilers are no exception, and modern ones are among the most complex pieces of software around. Making a useful compiler from scratch is no longer a weekend project–not without some serious help. Over the years, tools like bison and yacc[1] have sprung up to make parts of the job easier. More recent tools like LLVM[2] lower the barrier of entry to high performance compilation. There is sea of tools out there, and none are ever quite right for the job. Documentation is sparse, and the main

way to learn is by example. Tools must be used together, with lots of glue, to get the desired result: custom programming languages without compromise.

2.1 Motivation

The initial motivation for this research was the desire to revive an old language: Turing. Specifically, the Object Oriented version for Windows[3][4]. For many Ontario high school students, it holds a certain nostalgia and was likely the first programming language they learned. Wondering, "how hard could it be?" we set out to re-implement Turing. The answer to that question, as it turns out, is that it is pretty hard. The question shifted from "how hard" to "why is it so hard?" Identifying the difficulties and pain points of compiler construction might lead to betters tools for programmers and perhaps better productivity.

Not very long ago, Domain Specific Languages were written from the ground up to better fit specific problems sets. This way of doing things has gone out of practice, perhaps to blame is the friction involved with each step of building a compiler. It is likely straight-forward to build a compiler for one language for one architecture from scratch, but this approach throws out all the lessons that have been learned by mature compiler projects. To build a language that leverages decades of performance and security know-how without reinventing the wheel each and every time is more than desirable–with one catch: it should be easy.

3 Background

Programming languages are ubiquitous–without them, computers would not be the indispensable tools of endless capability that they are today. The fundamental idea of a programming language is to bring the ideas of computation closer to the language of humans. Doing so allows us to frame problems in ways that make sense to us, which can then be translated (or more accurately, compiled) into something a computer can process.

There are a wide variety of different programming languages for different tasks. Some languages are very high level and apply to a small problem space, and some are more generic for broad classes of problems. Like any good trade or craftsperson, programmers too have a set of tools to choose from. The onus is on them to pick the right tools for the job. Just as hand tools have evolved over time, so have programming tools. Of the tools a programmer reaches for most often, it is hard to overlook the compiler. Programmers turn ideas into code, and compilers turn code into work. It used to be that compilers were small hand written programs that read punched cards and spat out machine code. Now, they are among the most complex programs around. The complexity of a compiler comes from a desire for features, and a need for performance. Compilers look significantly different today from what they looked like on an IBM mainframe, but serve the same fundamental purpose: to make it easier to talk to a computer.

3.1 Inspiration and Prior Art

The ecosystem of compiler construction is huge. Many of the tools that are used to build a compiler, such as parser generators, are useful in other types of projects. In this environment, it is not really possible to find a perfect set of tools for a specific project. In practice this often means building things from scratch.

To evaluate which design decisions make sense for this new version of Turing, we looked to recent successful projects for inspiration. Even today, with the existence of large projects like LLVM, Antlr[5], and so on, there is no "right way" to make a new programming language. Everyone does it differently.

The main sources of inspiration are Go, Rust, and Zig[6].

Go is a modern outlier in that it makes no use (at least in the main project) of existing compiler infrastructure. At this point, the entire project is self-hosting. Everything from the parser to the assembler is written in Go[7]. At the beginning, however, it was bootstrapped with C.

Rust is a particularly interesting point of comparison in that it is built on top of the LLVM compiler infrastructure, which is a likely path for me to follow. How they bootstrapped their parser and lexer, and how they built their AST is an extremely useful source of inspiration and technique for this project.

Zig is a little out there and is also built on LLVM. As a small project without the backing of a big company, the decisions they made may be more relevant and applicable to this modern Turing endeavour.

Another potentially useful source of inspiration is the research project, Turing+. Naturally, as an implementation of the same base language, it should be useful to look at. Unfortunately most of parts interesting and applicable to this project are done with machine generated code and are not easy to study on their own. In particular it would have been useful to see how they implemented the parser and lexer. The project also lacks documentation and is written mostly in Turing itself. Parts of it may prove to be useful in the future after a deeper dive into the available source code. At the very least, it is another target to compare and test execution validity and performance against.

Other projects to look at include Clasp[8], Swift, ActionScript, Ruby, and Kaleidoscope[9]. Each of those languages has an LLVM front-end. Kaleidoscope is the official teaching example for how to get started with LLVM.

3.2 The Hard Way

Compilers have a long history of being written from scratch. Despite the availability of parser generators and compiler suites, today's languages reinvent the wheel in a lot of places. That is not to say this is a bad or unsuccessful approach. Go, for example, is a particularly good example of doing things from scratch. The initial compiler was written in C with a custom parser, pre-processor, and multi-target assembler. As the language matured and features were added, the compiler was ported to Go. This practice is called "self-hosting" and proves the viability and generality of a language.

Those looking to implement a high performance language, whether it is a domain specific language or general systems programming language, do not have the resources of Google and talent of grey-beard programmers behind them to implement an entire compiler stack from scratch. A programmer looking to implement a new language from scratch has a lot on their plate. At a low level, one must consider the ever changing variety of operating systems and processor architectures and how to optimize for each target. At a higher level, it is extremely easy to get bogged down in the details of parsing, lexing, and defining language semantics. All of this is on top of the job of designing a language that satisfies the user and offers something novel over existing languages. The task is daunting, to say the least.

4 tlang Case Study

tlang is an experimental implementation of The Turing Programming Language, which is a simple teaching language formerly used by some Canadian high schools. Turing was picked as a language to re-implement with modern tools for several reasons: 1. The language is well defined and well documented; no work needs to done to design a new language. 2. The language could use a new implementation. The language is abandoned and the current implementation is slow and only runs on 32-bit Windows. 3. It holds a special nostalgic place in the author's heart as the first language they learned. 4. The language's semantics are very simple and do not require a very complex run-time environment. For instance, there is no garbage collection.

Though Turing in its entirety was not implemented as part of this study, it served as an excellent foundation from which to learn about the difficulties and intricacies of programming language development. This strategy of picking an existing language (especially one that you already know) worked well. The more common approach is to implement a "toy" language with unrealistic or impractical semantics in order to learn how to use the tools. The problem is that once you achieve a basic understanding of these tools, it becomes hard to find additional examples of non-trivial use cases. To build a "real" language, once again the programmer is left on their own and must dive deep into highly technical codebases to gain further understanding. Although the LLVM Kaleidoscope tutorial was an indispensable resource for getting started with LLVM, it fell short as a reference for implementing some of the trickier features of Turing.

Tutorials for other tools are even worse, or non-existent. Those that do exist are usually based on the "classic" calculator example, usually implementing basic binary operations, and sometimes variables. Calculators have extremely simple grammars, and can easily be written by hand with around the same effort of setting up a parser generator and compiler toolchain.

4.1 The right tools for the job

The case study quickly highlighted one of the most challenging aspects of modern compiler construction: picking the right tools. Compiler construction is not as common of a task as it once was and is not very approachable for beginners, especially if they wish to produce a language with reasonable performance, perhaps within a factor of 10 of the speed of C.

The first (and some of the oldest) tools in the toolbox are lex and yacc, and their modern counterparts, flex and Bison. They are powerful, textbook parser generation tools that help programmers create all types of parsers, not limited to programming languages. The core pattern of programming with lex, yacc, and their derivatives is a deep connection with C. Even the most simple grammars contain snippets of C mixed in with BNF. These tools are hard to work with and carry legacy baggage with them, which gets in the way of productivity.

Experimenting with Flex and Bison led to little more than a small subset of the language being defined and no executable code was produced with this method. Frustration with these tools prompted the search for alternatives, which led to the discovery of ANTLR.

4.2 Transpilers

An honest and surprisingly common stopgap technique to leverage the power of existing compilers is to transliterate your language to another, most often C. This practice is called "transpilation." The benefits are clear: tons of library support, fast generated code, and plenty of target systems. The downsides are a little bit harder to identify. For some languages, the semantics of C may not offer neat mappings. How do you express the full generality of a Lisp variant in plain old procedural C? How do you guarantee the careful memory safety of Rust using only malloc and free? The reality of transpiled languages is that they inherit all the strengths and weakness of the target language. C was simply not designed with transpilation in mind.

If generating C is not the right choice, the only place to go is lower: to assembly language. Assembly language is a well traveled road, but if you're looking to go down this rabbit hole, there is a super highway to take instead: the LLVM Intermediate Representation (IR).

Also worth mentioning is C--[2], a subset of C designed as an intermediate language for high performance compilers. Sound familiar? This project has not seen much interest since the introduction of LLVM, but the sentiment definitely remains.

4.3 ANTLR

ANTLR is a parser generator that works alongside other compiler tools. It takes a BNF-like grammar and produces a parser program with simple object oriented patterns to hook into. Like most things in compilers, ANTLR is not new. Its predecessor, the Purdue Compiler Construction Tool Set (PCCTS) dates back

to 1989. However, ANTLR has been redesigned and built from the ground up a total of four times. To quote the primary author, Dr. Terrence Parr, "[it only] took 30 years to get it right!"

ANTLR offers something unique over tools like yacc and lex: it completely separates the grammar from pre-processing code. A single grammar definition file without any modifications can generate a high performance abstract syntax tree parser for C++, Java, Python, and Go.

Specific to the tlang case study, a complete grammar was written for Turing in under a day. Complete grammars for a wide variety of common programming languages are available in ANTLR's G4 format. The tlang grammar borrowed some of its structure from the C example in the ANTLR repositories. As it turns out, Turing shares a significant amount of structure with C with a few added niceties, which were easy to describe.

4.3.1 Example

This example highlights the best parts of making an ANTLR grammar.

```
integer_literal
    :   OCTAL_LITERAL
    |   sign=(PLUS | MINUS)? DECIMAL_LITERAL
    |   BINARY_LITERAL
    |   HEX_LITERAL;

OCTAL_LITERAL         :   OCTAL_PREFIX OCTAL_DIGIT+ ;
DECIMAL_LITERAL       :   DIGIT+;
HEX_LITERAL           :   HEX_PREFIX HEX_DIGIT+;
BINARY_LITERAL        :   BINARY_PREFIX ('0' | '1')+;
fragment DIGIT        :   [0-9];
fragment HEX_DIGIT    :   [0-9a-fA-F];
fragment OCTAL_DIGIT  :   [0-7];
fragment HEX_PREFIX   :   '0' [xX];
fragment OCTAL_PREFIX :   '0';
fragment BINARY_PREFIX : '0b';

PLUS    :   '+';
MINUS   :   '-';
```

A snippet of the tlang ANTLR grammar file

The above code is a grammar definition for several types of integer literals. The first statement is what is called a rule, in ANTLR. Rules can be nested within each other to create chains of tokens. All together, context emerges. From bottom to top, regular expressions match single characters as "fragments", which are then assembled into "tokens", like `DECIMAL_LITERAL`, which finally land as optional matches for the "rule" `integer_literal`. An example in the next section shows how an object representing this grammar snippet is used to create an LLVM `Value` object.

4.4 LLVM

LLVM is a relatively new player in the compiler space, first released in 2003. The LLVM Project is more than a replacement for GCC, it is an entire ecosystem for low level language development. Almost all of the significant new languages of the last decade are built on LLVM; the most noteworthy being Rust and Swift. The LLVM Project is also responsible for clang, a C and C++ compiler.

The LLVM ecosystem consists of several ideas that make it an excellent platform to build a modern language on. The modular structure of LLVM has a "write once; run anywhere" effect. The core technology that gives LLVM most of its power is the Intermediate Representation (IR) language. Instead of having to write a specific compiler for each target OS/architecture, you write a compiler (or transpiler) to the IR. IR code can then be compiled into any machine language for which an IR-to-assembly compiler exists. These concepts are referred to as "front-ends" and "back-ends." A huge weight is lifted off the shoulders of compiler writers everywhere, allowing programmers to focus on inventing new things. A perfect example of the power of LLVM was the introduction of the RISC-V architecture back end. Overnight (more or less), dozens of languages could target a brand new ISA after a single back-end was written. The same can not be said for any other compiler.

The IR is not limited to ahead-of-time compilation either. Part of the LLVM Project includes a just-in-time compiler, which is useful for dynamic languages.

The LLVM Project supplies a collection of C++ classes referred to as "bindings." Typical usage of them boils down to calling into them from your abstract syntax tree walker code which effectively builds a new abstract syntax tree in parallel in the context of LLVM IR. The C++ LLVM bindings do not provide a complete interface to all of the capabilities of raw LLVM IR code, but they are usable for most general tasks. Their primary advantage is abstracting the verbosity of raw LLVM IR into somewhat simpler C++ classes, which reduces the need to write an IR generator by hand. The disadvantage, besides incomplete coverage of all of the features of LLVM, is their verbosity and lack of flexibility. Though C++ is a very flexible language, the API bindings are very strict in how they are used. A task such as expressing the addition of two scalar number types is about as simple as it gets, and requires a significant amount of boiler plate to accomplish. The bindings also lack helpers for generating higher level constructs, like virtual function tables, and to do so you must drop down to LLVM IR verbosity.

The bindings are good place to start for implementing a simple language on LLVM, but are not generally used for practical compiler work. Languages like Rust and Swift generate IR code directly, for example.

4.4.1 Example

This example shows a simple work flow for handling the objects an ANTLR parser creates. The end result is the creation of an LLVM `Value` object, one of the core IRBuilder classes.

```
llvm::Value* handleIntLiteral(tlangParser::Integer_literalContext *l,
    const char* str, int base, bool is_signed) {
    char *strtolchar;
    uint64_t i = strtoull(str, &strtolchar, base);

    if (errno == ERANGE) {
        printErr(l->getStart()->getLine(),
            l->getStart()->getCharPositionInLine(),
            "semantic", "integer␣literal␣too␣big");
    }
    return llvm::ConstantInt::get(TheContext, llvm::APInt(64, i,
        is_signed));
}

llvm::Value* handleLiteral(tlangParser::LiteralContext *ctx) {
    char *strtolchar;
    if (ctx->integer_literal()) {
        auto l = ctx->integer_literal();
        if (l->DECIMAL_LITERAL()) {
            if (l->sign) {
                // signed integer
                if (l->sign->getType() == tlangParser::MINUS) {
                    return handleIntLiteral(l, l->getText().c_str(), 10,
                        true);
                }
            } else {
                // unsigned integer
                return handleIntLiteral(l, l->getText().c_str(), 10,
                    false);
            }
        } else if (l->HEX_LITERAL()) {
            // shift start of string by length of prefix
            return handleIntLiteral(l,
                l->HEX_LITERAL()->getText().c_str() + 2, 16, false);
        } else if (l->OCTAL_LITERAL()) {
            return handleIntLiteral(l,
                l->OCTAL_LITERAL()->getText().c_str() + 1, 8, false);
        } else if (l->BINARY_LITERAL()) {
            return handleIntLiteral(l,
                l->BINARY_LITERAL()->getText().c_str() + 2, 2, false);
        }
    }
    return nullptr;
}
```

A C++ snippet from tlang's compiler

The previous code sample "jumps in" at a specific point in the ANTLR parse tree and walks the rest of the tree's children manually. A separate helper function, `handleIntLiteral`, builds the appropriate node object for IRBuilder to include in the generated IR code. This code forms part of the "pre-processor." The program that runs in this step only produces values that will be included in the resulting code. It is also the place that was chosen to handle a specific type of error. ANTLR has given us an object that holds information like the line and character number which lets us generate helpful errors.

4.5 GCC

Conventional compilers, the most notable being GCC, do not lend themselves to extension as easily as other modern tools do. Deep understanding of GCC's code base is required to add new language support to take advantage of its optimization layers. Over the years, language support has been added to GCC that use its internal "middle" representation. There was a time where the performance of GCC code was unmatched, but LLVM has caught up and begins to exceed GCC in certain workloads. As a platform for creating a new programming language, GCC is not a practical approach.

5 The Missing Tool

An outcome of the tlang case study was the identification of several shortcomings in the current compiler tooling ecosystem. Even with the immense productivity increase from ANTLR and LLVM, truly seamless language construction is beyond reach. This section outlines some ideas of what a bridge between a parser generator and intermediate language might look like. ANTLR and LLVM respectively will be used as the book-ends, so to speak, of this proposal language.

5.1 Implementing tlang

The first experiments to get tlang off the ground were with Flex and Bison. With a little effort it was quite easy to write a parser generator for the binary arithmetic portions of the Turing language. Beyond that, circular ambiguities in the BNF grammar were difficult to avoid, and the reliance on C union type structs proved to be confusing, brittle, and hard to build on. For example, it was straightforward to carry the value of a keyword in the union, as it was represented as a unique number. It was also easy to carry a raw integer or floating point value in the union as C allows for blind casting between types. When it came time to encode variable length string literals and things like type identifiers, ugly hacks were required to keep track of where those values were stored and how they ended up at their destination. The available literature focuses on building general parsers, usually for markup languages or configuration files, and does not offer much to those looking to build a programming language, especially those looking to use LLVM. One more thing to note on yacc/lex and

their derivatives is a lack of consistency on where to put pre-processing and processing code. Again, the literature does not make this clear, as many approaches are seen as valid. Bison and Flex are powerful, high performance tools, but are not approachable for beginners.

Not long after the experiment with Flex and Bison, a half-hearted attempt to write a parser by hand was done in Go. It used a recursive descent parsing strategy that worked well for a subset of the language. The construction of the parser did not have a satisfactory way to handle complex grammars and reached a point where it needed to be rewritten.

Before starting over once again and rewriting the parser by hand, a fair shot was given to ANTLR. ANTLR was initially looked over to avoid introducing a Java dependency to the project. However, it was quickly shown that any distaste for Java could be overlooked due to the gain in productivity it provides.

The most successful implementation of tlang was a simple combination of the tools previously discussed. It started with an ANTLR grammar. It was used to generate a simple visitor pattern interface to ANTLR parse tree. The C++ code it generated, was used as a template to insert LLVM's C++ bindings into, which then generated IR code. The reality of this process was that it inevitably led to tightly bound, hard to maintain code that offered very little flexibility. ANTLR itself was not constricting, but it did not mesh particularly well with the LLVM IRBuilder API. LLVM's bindings were found to be extraordinarily dense and difficult to understand. Most of the successfully implemented features were those covered by the Kaleidoscope tutorial. The primary issue found with IRBuilder was that the "official" documentation entirely consists of machine generated documents. Perhaps they are useful for internal reference, but to a set of fresh eyes, they are completely unapproachable. The closest thing to "documentation" LLVM provides is the Kaleidoscope tutorial. The tutorial has a fairly low barrier of entry, but unfortunately only gives a superficial overview of how to use the LLVM bindings. Kaleidoscope's lack of depth perhaps stems from its use of a hand-written parser without properly defined language specification. It is understandable that the writers may have wished to keep the scope of the tutorial narrow by avoiding the inclusion of some sort of parser generator, but it resulted in a less than accurate picture of what compiler development requires in full.

5.2 The Semantics Problem

The definition of programming language semantics happens in two major stages.

1. Pre-processing: Converting a token stream into usable data for processing. If "4 + 4" is run in this step, "8" ends up in the final executable.

2. Processing: Take the pre-processed data to define operations. If "4 + 4" is run in this step, "4 + 4" ends up in the final executable.

The stage in which a language feature falls, has some effect on how dynamic a language is. In a language like C, all types are checked and reduced to their most primitive form by the pre-processor and are used directly by the computer. In a language like Ruby, the type system is an abstraction layer on top of primitive types where actual types are only resolved at runtime, meaning that additional helper code must be included to help with these tasks.

The LLVM IRBuilder classes care only about the "processing" stage, as they likely should. Naturally, this means that the programmer is left to write their own pre-processing stage. For small, or toy languages, the pre-processor code can exist pretty much anywhere you want. It can live with the parser generator, which is a common strategy in the lex/yacc world; it can stand on its own, much like the C pre-processor; or it can live with the code generator, a simple solution when using the LLVM IRBuilder classes.

5.3 Ideas

A tool is missing to bridge the gap between tools like ANTLR and LLVM. What we need is an easier and methodical way to define language semantics in the context of LLVM in a way that integrates with a BNF grammar. The current language for doing so is a stiff C++ API library, or manually generating LLVM IR by hand. The solution has to be a piece of software: perhaps the solution could be a programming language of its own.

5.4 A Programming Language for Writing Programming Languages

A wish list of features for such a language would be roughly as follows:

- Provide simple and concise syntax for defining arbitrarily abstract assemblies of programming primitives
- Have a focus on productivity
- Maintain full (or near full) expressiveness
- Integrate seamlessly with a parser generator
- Provide first class support for all programming paradigms
- Provide resources for defining higher level features like concurrency, lambdas, file I/O, memory protection, etc.

Some suspected consequences and limitations might be:

- Tight integration with specific tools
- An opinionated way of doing things
- Lower performance than doing things by hand

6 Conclusion

Language construction has come a long way since the days of punched card programming. Tools like ANTLR give us the power to create complex abstract syntax tree crawlers in many different languages from simple grammar files. LLVM gives us the power to produce high performance code for countless machines and operating systems with its Intermediate Representation. Still is lacking is a way to bridge these powerful tools together.

References

[1] John R Levine et al. *Lex & yacc*. O'Reilly Media, Inc., 1992.

[2] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. "C--: A Portable Assembly Language that Supports Garbage Collection". In: *Principles and Practice of Declarative Programming*. Ed. by Gopalan Nadathur. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 1–28. ISBN: 978-3-540-48164-5.

[3] info@compsci.ca. *Turing for Windows*. URL: `http://compsci.ca/holtsoft/`.

[4] *Turing Documentation*. URL: `http://compsci.ca/holtsoft/doc/`.

[5] Terrence Parr. *ANTLR*. URL: `https://www.antlr.org/`.

[6] *The Zig Programming Language*. URL: `https://ziglang.org/`.

[7] *Go 1.6 Release Notes*. Feb. 2016. URL: `https://golang.org/doc/go1.6`.

[8] Christian Schafmeister. *Clasp: Common Lisp using LLVM and C for Molecular Metaprogramming*. June 2015. URL: `https://www.youtube.com/watch?v=8X69_42Mj-g`.

[9] *Kaleidoscope: Implementing a Language with LLVM*. URL: `https://llvm.org/docs/tutorial/LangImpl01.html`.