---
layout: post
title: "Writing A Compiler"
date: TODO
date: 2020-10-07
categories: code

At the start of the 2020 school year I wanted to learn more about compilers so I started writing a compiler for the ez programming language that I made up. I did this as an independent study for school. I wanted to grow *dramatically* as a thinker and learn a lot about computer science and compilers in specific. If you just want to see the project head to github.com/g-w1/ezc or g-w1.github.com/ezc for documentation.

> Note: I may make some mistakes in this post. If that happens, please send me an email and I'll correct it. The code examples in this post were taken from an earlier stage of my compiler and are simplified a little.

Why I Chose Rust

I chose Rust for this because I wanted to learn it and it is just a very cool language. If you don't know of it, I think of it as aiming to be a replacement to C++. Some language features that make it very easy to write a compiler in are:

Enums (algebraic data types)

Algebraic data types are most present in functional languages (haskell, ocaml, etc).

These allow for efficient representation of tokens and nodes of an abstract syntax tree in memory. In c one could think of them as tagged unions. For example my Expression type is defined as this

```
pub enum Expr {
    /// an identifier
    Iden(String),
    /// a binop
    BinOp {
        lhs: Box<Expr>,
        op: BinOp,
        rhs: Box<Expr>,
    },
}
```

This means an expression is either a number, an identifier (variable name), or a combination of 2 expressions with a Binary operator in between. It is defined

as Box because recursive data types are not allowed, so it has to be a pointer (a box is basically a pointer to the heap)

Pattern Matching

Along with algebraic data types, pattern matching comes. It is used to destructure an enum into the parts that it is made up of. It is best explained with an example:

```
match e { // e is the expression we are matching against
  Expr::Number(n) => println!("It is a number: {}", n),
  Expr::Iden(i) => println!("It is an identifier: {}",i),
  // This will capture everything and put it in a fallback variable. We know is has to be a
  recurse => panic!("Uh-oh we do not know how to do recursive expressions yet: {:?}", recurs
}
```

As you can imagine, Rust enums are **so** cool. Most of the higher level types in my compiler are enums compared to structs. I can't imagine doing this in a language like c where I would have to use many layers of tagged unions and structs to achieve the same.

Enums and Pattern Matching were the main reasons I chose Rust over something like Python. Another reason is probably the strong typing. It makes it soo much easier to debug your program when you know exactly what each function will return.

The last reason I chose Rust over a language like c++ is because of the safety guarantees. It makes your life easier when you can debug why the assembly code your compiler is generating is segfaulting rather than why the compiler is segfaulting .

Structure

The structure of my compiler looks like this:

```
Parse Commands -> Lexer -> Parser -> Semantic Analyzer -> Code Generator -> Write To File ->
```

Parse Commands

The command line parsing is very simple. I just use `std::env::args` in Rust to get a `Vec<String>`. I just use the 1st arg to get the file to compile and then just use a loop to get the rest. It is pretty easy.

Lexing

The Lexer is next. It takes the source code and breaks it up into tokens. Each token is an option in a `Token` enum. The definition looks something like this

```
pub enum Token {
    // Keyword Tokens
    /// kword for set
```

```
    Kset,
    /// kword for Change
    Kchange,
    /// to
    Kto,
    /// If
    Kif,
    /// Loop
    Kloop,
    /// break
    Kbreak,
    /// Function
    Kfunc,
    /// Return
    Kreturn,
    // Iden tokens
    /// Identifier token
    Iden(String),
    /// IntLit token
    IntLit(String),
    /// EndOfLine token (.)
    EndOfLine,
}
```

The way the lexer works is through a state machine: The states are represented by this enum:

```
enum LexerState {
    Start,
    InWord,
    InNum,
    SawLessThan,
    SawEquals,
    SawGreaterThan,
    SawBang,
}
```

Then to lex I just iterate through all the characters in the input string and match to them with the state. If I see a `'<'` and am in the `Start` state, then I will set the state to `SawLessThan`. Then if I see an `'='`, I will append a `GreaterEquals` token and set the state to start, and if I see anything else, I will append a `GreaterThan` and set the state to start and go one character back. If I am in the `InNum` state and see a number then I will append it to the intermediate string and keep going. If I see something else then I will set the character back one and set the state to `Start`. I think you get the gist of this approach.

To get an idea of even *how* to write a Tokenizer/lexer I started reading the source code of the zig programming language (it is aimed to be a replacement to c).

The codebase is in c++/zig which is similar enough to Rust to understand. It is also relatively small. It is in just one flat directory with around 30 files. I found this much easier to navigate than the Rust codebase, which has *tons* of directories. I would recommend the zig language for learning how to write a language.

Parsing

The Parser is a little more complicated than the Lexer. It is a recursive descent parser, which means that it can call itself. I wrote small helper functions to do small, repetitive tasks in the compiler, and then functions to parse different types of `Ast` elements. This is an example of the function that parses a function.

```rust
/// Function <- FnProto OpenBlock Ast CloseBlock
fn parse_func(&mut self, tree: &mut Vec<AstNode>) -> Result<(), ParserError> {
    self.expect_eat_token(Token::Kfunc)?;
    let (name, args) = self.parse_func_proto()?;
    self.expect_eat_token(Token::Comma)?;
    let body = self.parse(false)?;
    self.expect_eat_token(Token::ExclaimMark)?;
    tree.push(AstNode::Func {
        name,
        args,
        body,
        vars_declared: None,
    });
    Ok(())
}
```

You can see it calls the parse function. The arg it takes is a `bool` that tells it whether it is top-level of not. We call it with false here because it is parsing inside the function. It also calls `parse_func_proto`, another helper function.

Another reason why I like Rust is because of it's error system. The question mark `?` operator is used for a function that can return an error. You can kind of think of it like a monad binding in Haskell except now it only supports `Option` and `Result`. If the function returns an error it will be passed along, but if the function does not return an error, then the error will be unwrapped. The error type in rust is `Result<T, E>`; it is an Enum. The question mark is just syntactic sugar for

```rust
let unwraped_res = match func_that_returns_result() {
  Ok(o) => o,
  e => return e,
}
```

The rest of the parser is fairly straightforward. I just have a giant `match` statement that matches on all types of statements and then dispatches to the function that handles that type of statement.

Semantic Analysis

This part was not very hard to implement, but it was *very* fun. I have it as an analyzer struct:

```
struct Analyser {
    /// the initialized_static_vars
    initialized_static_vars: HashSet<String>,
    /// the initialized_local_vars
    initialized_local_vars: HashMap<String, u32>,
    /// the initialized_function_names
    initialized_function_names: HashSet<String>,
    initialized_function_vars: HashMap<String, u32>,
    /// scope that the analizer is in rn
    scope: Scope,
}
```

I make ample use of `std::collections::HashMap` and `HashSet` in the analyzer and the code generator. The latter is just a wrapper over `HashMap<T, ()>` for using a set with near constant time lookup whether something is in it.

This captures all of the variables that are initialized in every scope and a scope struct that looks like this:

```
struct Scope {
    in_if: bool,
    in_loop: bool,
    in_func: bool,
}
```

From here, I just have to recursively descend the Ast in a form of a `Vec<AstNode>` that is recursively traversed by `Analyser::analyze` on `Analyser`.

Here is an example method in `Analyser`:

```
/// a helper function to make sure a variable exists
fn make_sure_var_exists(&self, var: &str) -> Result<(), AnalysisError> {
    if !self.scope.in_func {
        if !self.initialized_local_vars.contains_key(var)
            && !self.initialized_static_vars.contains(var)
        {
            return Err(AnalysisError::VarNotExist(var.to_owned()));
        }
    } else {
        if !self.initialized_function_vars.contains_key(var) {
            return Err(AnalysisError::VarNotExist(var.to_owned()));
        }
    }
```

```
    Ok(())
}
```

The analyzer also modifies some of the ast nodes and adds the variables declared in them to them. This helps the code generator. I did not use an intermediate representation since I am not doing any code optimisation, but it seems *really* interesting. Ill definitely look into it in the future. Maybe for zig's self-hosted compiler. I actually came up with the exact same method of adding `Option<_>` fields to the Ast that would be mutated by the Analyser as Tristan Hume did when writing his compiler without even knowing it! His blog post actually was the original thing that inspired my interest in compilers, but I didn't understand a lot of it. Now I do!

Code Generation

This was the hardest part by far to implement. That will make it more fun to talk about!

I have a struct `Code` that tries to mirror assembly with sections. It has `text` and `bss`. These each have `instructions` which is a `Vec<String>`. In hindsight, it would have been simpler to just have a `String` instead of `Vec<String>` for storing the code.

> Note about optimisation: In this compiler, I am not worried about optimisation of the *compiler*. This is why I use types like `String` and `Vec` that are heap allocated a lot. The compiler is still pretty fast, I assume this is because the lack of optimisation in the generated code and the fact that it is ~1.3 pass (I count sorting the Ast as .2 and removing `_start` if compiling for a library as .1).

Then I just loop over the Ast and do code generation. The part that I found the *most* interesting was stack allocation. It took me about a week to think of an algorithm to impliment this. First of all, stack allocation is used where the data is deallocated after it goes out of scope (when the function or if statement ends). It is actually so difficult that I have to keep track of the stack pointer offset in code. The general algorithm looks something like this:

1. At the time of allocation, insert into a `HashMap<String, u32>` (lets call it `initalized_vars`) the name of the var, current stack pointer - the place of the var that is initialized in all the vars that were initialized in that block.

```
self.initalized_vars.insert(varname, self.stack_p_offset - place);
```

2. To find the offset of that variable from the current stack pointer just look up in `initalized_vars` the variable name and take the value and subtract it from the current stack pointer to get the offset from the current stack pointer.

```
self.stack_p_offset - self.initalized_vars.get(varname);
```

Yeah, that algorithm took me a *really* long time to come up with, but it payed off in the end. If you have a better one please tell me :). I have taken I deeper look at the codegen in the zig programming language and it seems like they use this same algorithm, so I think its good. When implementing arrays, I used a slight variation on this algorithm.

Since I also support writing libraries, I have an option to filter out `_start` when generating the code. Here is an example method to generate code for an expression:

```
/// code generation for an expr. moves the result to r8
fn cgen_expr(&mut self, expr: Expr) {
    match expr {
        Expr::BinOp { lhs, op, rhs } => {
            self.cgen_binop_expr(*lhs, op, *rhs);
            self.text.instructions.push(String::from("pop r8"));
            // we decriment the stack_p_offset by 1 because we pop off the stack
            self.stack_p_offset -= 1;
        }
        Expr::Number(n) => self.text.instructions.push(format!("mov r8, {}", n)),
        Expr::Iden(i) => {
            let r = self.cgen_get_display_asm(&Expr::Iden(i));
            self.text.instructions.push(format!("mov r8, {}", r));
        }

        Expr::FuncCall {
            func_name,
            args,
            external,
        } => {
            self.cgen_funcall_expr(&func_name, external.unwrap_or(true), &args);
        }
        Expr::DerefPtr(a) => {
            let r = self.cgen_get_display_asm(&Expr::Iden(a));
            self.text.instructions.push(format!("mov r8, {}", r));
            self.text.instructions.push(format!("mov r8, [r8]",));
        }
        Expr::AccessArray(a, e) => {
            self.cgen_access_array(&a, &*e, true);
        }
    }
}
```

Write To File and Link

This was also very easy. I just used `std::fs::write()` to write the assembly and then used `std::process::Command` to run `nasm` and `ld`.

Standard Library

No programming language is complete without a standard library. To implement the standard library of ez, I used the zig programming language. I chose this for a few reasons: it is very easy to compile a library with zig, and I wanted to learn zig. Now I am contributing to the zig compiler! The only challenging thing was integrating with how arrays work in ez. The memory layout of arrays is slightly different. You can see the standard library here.

What I learned from this

I am really glad I chose to write a compiler (and standard library). It took ~50 hours, but it was worth it. I learned so much about programming.

Here is a list of things I learned:

- Testing: I have not done very much testing in previous projects so in this project I did. I'm not really sure what type of testing it was. I tested the compiler and the stdout from the generated code. The compiler tests were testing the generated data structures for the lexer and parser and regression tests for the analyzer and code generator. I implemented the stdout tests with 2 scripts. `gen_output.sh` would run every .ez file in the tests directory and store its output. `test.sh` would run every .ez file in the tests directory and check it against what `gen_output.sh` generated. Even if I didn't catch many bugs with all this testing, it felt satisfying to see ~60 tests passing! I also setup Github Actions and had it run the tests on any pushes to master. I ran into a weird bug with nasm, my assembler, in which the feature set on Ubuntu (github actions) was different than on Arch Linux (my distro). I had to dumb down some code so that it would also run on Ubuntu.
- Recursion: I had known about recursion before writing this compiler, and had implemented some recursive algorithms, but I felt this really cemented the idea in my head. With 2 of the 4 main compiler steps using a recursive algorithm (parsing and analyzing) I feel like I have a very good understanding of recursion. I also discovered a pattern with recursive methods on a struct. This way, data can be shared between methods without passing it between functions in a cumbersome way.
- General Software Design: I feel like I have just learned a lot about designing software from this project. It is biggest project I have made and I am pretty happy with the design. The code for lexing, parsing, and analyzing is pretty nice. The code quality goes downhill when you get to `codegen.rs`. I attribute this to not using recursion: this method requires a lot of repeated code that could be avoided with recursion.
- Low Level Programming: I learned about linking, assembling, calling conventions, and much more.

What's Next

My journey about learning about compilers is just starting! I have so much to learn. From the time I was done with ez to the time this blog post was published and onwards I have started contributing to zig and related projects. I can see myself doing this for a while longer. I have found that reading high quality code is a very good method of learning. I eventually want to learn about operating systems and the linux kernel and will have to learn c or c++ eventually :(. I might want to write a shell in c to learn about linux syscalls. I also want to take ap computer science soon, so will have to learn Java :(. Hopefully, I come to like these languages even though I don't think I will. Until my next blog post, keep learning!