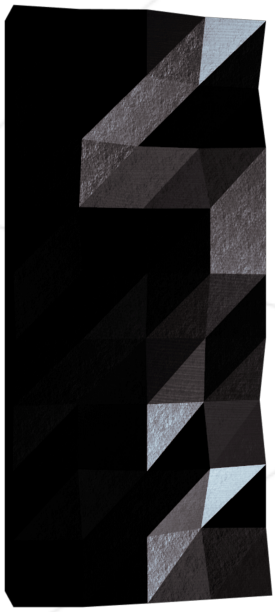


cc1

For front-end developers

Made with ❤️ by Citron 🍋

Summary: This project is about creating a C compiler

Version: 1.00

Contents

I Objective **3**
I.1 Example 5

II General rules **6**

III Mandatory part **8**
III.1 The program 8
III.2 The driver 9

IV Bonus part **10**
IV.1 Cross compilation 10
IV.2 Metaprogramming 10
IV.3 Debugging 10

V Resources and references **11**

VI Submission and peer-evaluation **12**

I lied. I don't have Netflix.

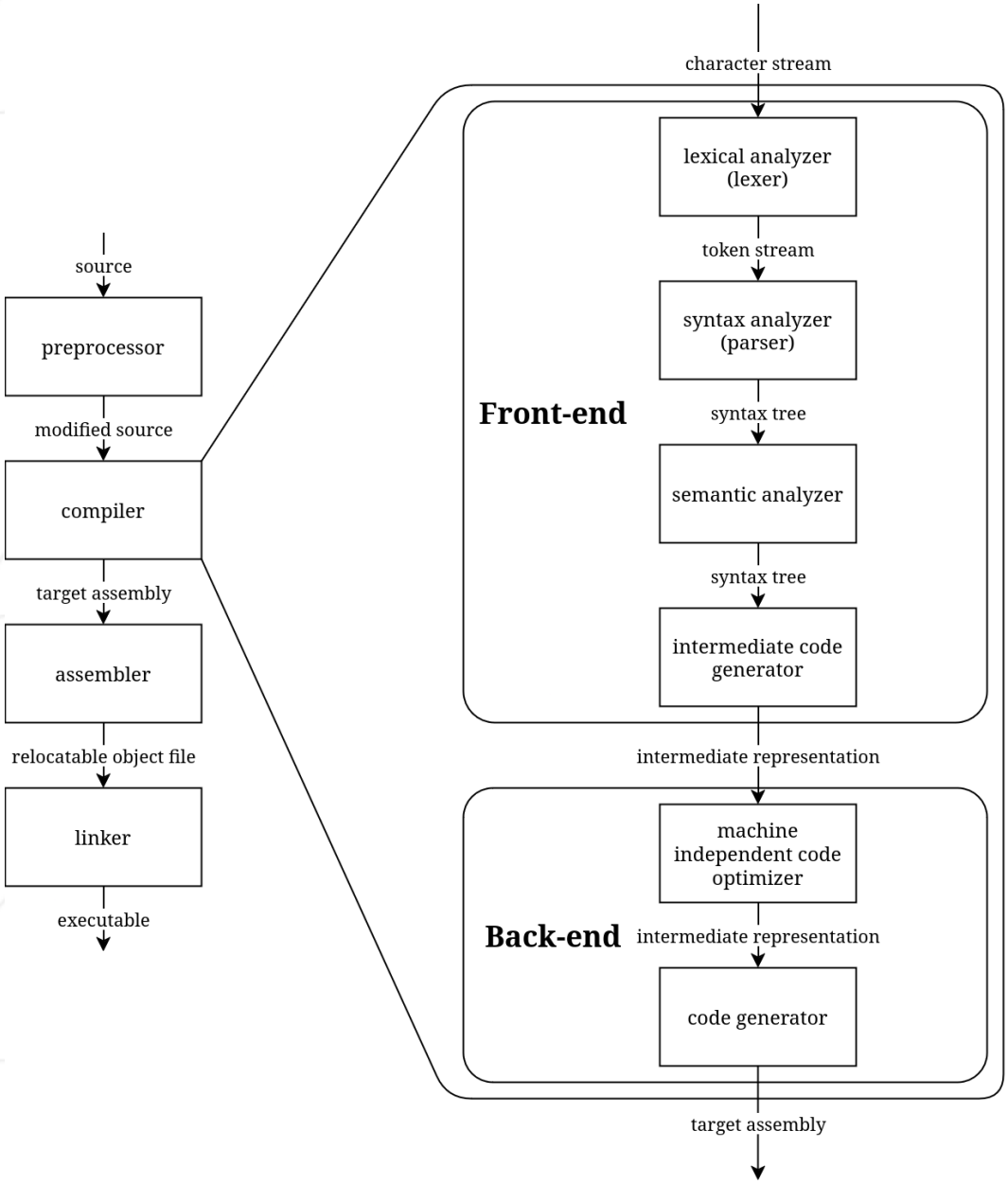
Take off your shoes and get comfortable.

We're going to design and implement a complete C to x86_64 compiler from scratch

Chapter I

Objective

Take a look at the following diagram:



On the left side, we see the **compiler toolchain**, a set of tools (e.g., gcc, as, ld...) that are executed each time a file is compiled. These tools are executed automatically and managed by the **compiler driver**. Most of the time, when you use a compiler like gcc or clang, you are actually using a compiler driver. The compilation process itself doesn't occur within the tool but is handled by other programs called in sequence by the driver. A good way to understand what happens when you use a compiler driver is to run the command `gcc -v <cfile>`.

The real compiler (also called **compiler proper**) is the program that takes C source code and translates it into assembly language (this is what you did in B). This process is actually quite complex and can be divided into several phases, as illustrated on the right side of the diagram.

In the previous project (B), the language was so simple that you could simplify the compilation process to just three steps: **Lexing**, **Parsing** and **Target code generation**.
The main difference between B and C is the **type system**. The fact that B has no types makes the language much simpler, removing the need for semantic analysis. In B, expressions are always semantically correct because there are no type constraints, and **l-values** are distinguished from **r-values** at a **syntactic level**. In fact, the only **semantic** check needed is the verification for **symbol** existence. Additionally, there was no need for **intermediate representation** as every instance of an operator translated to the same target code (e.g., in B a `/` always translates to `idivw` but in C, it can be `divss`, `divsd`, `idivl`, `divw`...), we only targeted i386 and we didn't want to optimize the generated code.

In a well-designed C compiler, all compilation phases are necessary. These phases can be grouped in two parts:

- The **front-end** responsible for analyzing the source language.
- The **back-end** responsible for generating code in the target language.

Between these two parts, we introduce the concept of **Intermediate Representation (IR)**. Strictly speaking, an IR is any representation of the program throughout the compilation process (token list, AST...) but when we talk about **the** intermediate representation, we mean a specific representation (often textual) that is high-level enough for control-flow optimization while being low-level enough to represent various programming languages.

This interface allows us to place the IR at the core of the compilation process so that we just need to create **one front-end per programming language** and **one backend per architecture** enabling any language to work on any architecture.

The concept of **middle-end** is sometimes used to describe components that take as input and give as output intermediate representation in order, for example, to perform machine independent optimization.

This project involves creating **a C front-end for LLVM**.

And of course, standard-compliant (as always).

I.1 Example

```
$> cat main.c
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
$> ./cc1 main.c -o-
target datalayout = "i8:8:8-i16:16:16-i32:32:32-i64:64:32-f32:32:32-f64:64:32-p32:32:32"
target triple = "i386-redhat-kfs"

declare i32 @printf(ptr, ...)

@str = private global [14 x i8] [i8 104, i8 101, i8 108, i8 108, i8 111, i8 44, i8 32, i8 119, i8 111, i8
     114, i8 108, i8 100, i8 10, i8 0]

define i32 @main()
{
0:
    %1 = call i32 @printf(ptr @str)
    ret i32 0
}
```

Chapter II

General rules

- This project will only be evaluated by actual human beings. You are therefore free to organize and name your files as you wish, although you need to respect some requirements listed below.
- You are allowed to write your program in any of the following languages:
  - C
  - C++
  - Java
  - Zig
  - Rust
  - Caml/OCaml
  - Any **purely functional** language
  - assembly (please don't)
- In all cases, you are only allowed to use the standard library.
- If you use a compiled language, you must submit a Makefile or a configuration for the native build system of your language.
- All the code present on the repo must be yours.
- You are not allowed to use a code generator (such as yacc or lex) that you haven't implemented yourself.
- You are never allowed to include generated code in your repo, if you use ft_lex or ft_yacc you must commit the .l and .y files and your makefile or build config must call the generators automatically.
- If you use your own ft_lex or ft_yacc you must include the sources as a subdirectory and your makefile or build config must compile it.
- You must handle errors in a sensitive manner. In no way can your program quit in an unexpected manner (Segmentation fault, bus error, double free, etc).

- If a user error occurs (eg: duplicate variable definition), your program must provide a detailed error (eg: `main.c:42:1 redefinition of 'var'`)

Chapter III

Mandatory part

III.1 The program

- You must implement a C compiler.
- The program must be named **cc1**.
- The **synopsis** of this program must be `cc1 infile [-o outfile]`.
- The input file contains C code.
- The output file must contain the **LLVM intermediate representation** corresponding to the source code.
- You must use one of the latest versions of LLVM.
- The **target architecture** is **Intel386**.
- The C version must be **C89** (ANSI/ISO 9899-1990).
- You can implement features from other versions of C, but you must support all the features from C89.
- You must read and comply to the standard.
- You are not required to handle **preprocessing** (translation phases 1 to 4 in section 5.1.1.2 of the standard) in the mandatory part.
- You are not required to implement the **standard library**.
- As a consequence of the previous requirement, you don't have to handle **va_args** in function definitions, but you must handle them in function calls (printf will be used during the evaluation).
- You must comply with the **system V ABI** for the target architecture.

We advise you to use your previous work (lex and yacc) in this project but this is not mandatory. You are not obliged to do a syntax directed translator, for example you could use a recursive descent parser or combinator parsing.

III.2 The driver

You must submit a **compiler driver** along with your compiler, this program will use your compiler along with tools like `as` and call the entire compilation chain and enable you to use your compiler similarly to `gcc` or `clang`.

- You must implement a compiler driver.
- Your driver must be called **fcc** (yes the Forty-two Compiler Collection).
- You can implement this driver in **any** language (C, C++, JavaScript, Python... (Shell is a good choice))
- Your driver must execute the following steps:
  - C preprocessing using the `clang -E -std=c89` command
  - C to LLVM compilation using your `cc1`
  - LLVM to assembly lowering using the `llc` command
  - assembly to ELF using the `as -c` command
  - ELF linking using the `clang -m32` command (since this project doesn't include any standard library or runtime, we will use the runtime and library of clang, this is why we let clang do the linking)

  (you may add options to these commands)
- Your driver must provide the same interface as the `POSIX.1-2024` `c17` utility following these sections:
  SYNOPSIS, DESCRIPTION, OPTIONS, OPERANDS, STDIN, INPUT FILES, STDOUT, STDERR, EXIT STATUS
- You must implement all the options of `c17` except `-g -B -G -R`.
- the `-I -D -U` options can be passed directly to the preprocessor, the `-L -l -s` options can be passed directly to the linker, the `-O` option can be passed directly to the code generator (llc).

`Imagine being one of the most influential anarchists of the late 20th century and being known in computer science for creating a hierarchy. (nothing to do with this subject, just wanted to put that somewhere)`

Chapter IV

Bonus part

IV.1 Cross compilation

Add support for x86_64. All aspects of your program must be consistent, including structure offsets, alignments, and constant expression evaluations. for example, considering the following enum:

```
enum e {
    A = ~(unsigned long int)1 % 7
};
```

A is a compile-time constant. On i386, `A` would be evaluated to `2`, but on x86_64 `A` would be evaluated to `0`, as `unsigned long int` has different representations in the two architectures.
You must add an option to your compiler and your driver to switch between architectures (eg: default to x86_64 and `-m32` to switch to i386).

IV.2 Metaprogramming

Implement a preprocessor, either as a separate program or integrated in the main program.
It must perform translation phases 1 through 4 (see section 5.1.1.2 of the standard).
It must replace the `clang -E` command in your driver.

IV.3 Debugging

Add support for the `-g` flag, when this option is enabled, the output of your program must include detailed LLVM debug information that will allow you to use gdb on the output executable.

The bonus part will only be assessed if the mandatory part is PERFECT. Perfect means the mandatory part has been integrally done and works without malfunctioning. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.

Chapter V

Resources and references

- Aho, Lam, Sethi, and Ullman, ***Compilers: Principles, Techniques, and Tools*** (aka the dragon book), Pearson Education.
  Often referred to as the bible of compilers, this book provides comprehensive coverage of compiler architecture and principles.
- Levine, Brown, and Mason, ***lex & yacc***, O'Reilly.
  Implementing lex and yacc is only part of the challenge, this book serves as a foundational guide to using them efficiently and effectively to build a strong parser.
- Peter van der Linden, ***Expert C Programming: Deep C Secrets***, Pearson Education.
  This book contains a lot of anecdotes on C history and provides insight into advanced C concepts, I strongly recommend you to read the section "`Reading the ANSI C Standard for Fun, Pleasure, and Profit`".
- godbolt.org. This website offers a large collection of online compilers, allowing you to quickly observe how C constructs translate to LLVM IR and other assembly representations.

Chapter VI

Submission and peer-evaluation

Submit your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your folders and files to ensure they are correct.