Compilers

-Tadi Joshua Raj

What is a compiler?

- It is a tool which translates a high-level language into a low-level language.
- Now this brings few questions to mind:
  1. Is that all a compiler does?
  2. Why do we need to use a compiler?
  3. What is an interpreter? does it also translate?
  4. By *translate* can we change ***C*** into ***JavaScript***? Or is it only limited to HLL -> LLL?

Why we use a compiler:

- In the early 1950s, there were no programming languages, and all code had to be written directly in machine code!
- We needed a more efficient way to interact with computers.
- The solution was to write code in a higher-level programming language, which could then be translated into machine code using a compiler.

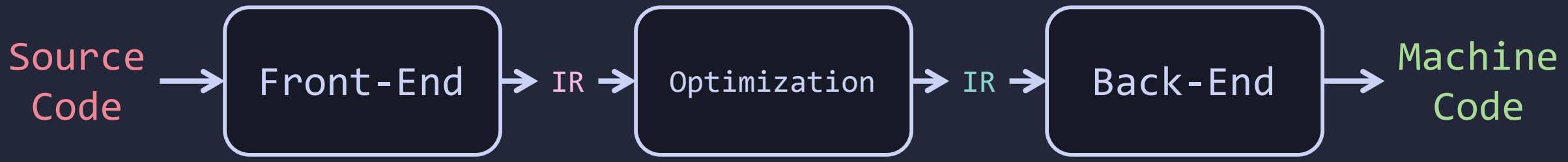
History of Compilers:

- In 1957 the first programming language FORTRAN (FORmula TRANslation) was invented, its compiler translated complex mathematical formulae into efficient machine code.
- In 1958 LISP's compiler introduced dynamic memory allocation and garbage collection.
- In 1959 COBOL's compiler focused on portability, allowing the same code to run on different machines.

- In 1960 ALGOL's compiler introduced structured programming and syntax trees, which became critical for modern compiler design.
- In 1972 C programming language was developed, requiring compilers to optimize for performance.
- Coming to the modern era, we have made significant advancements in the compiler industry, these are some of the modern compiler used:
  - Just in time compiler (more on this later)
  - Cross-platform compilers and so on
- Some modern compilers even integrate AI to predict optimization strategies and improve performance.

A modern compiler:

- There are three phases:



- Let’s look at them briefly.

- Front End:
  - Converts the code into an intermediate representation
  - It involves preprocessing, tokenizing, parsing, and then finally generating the IR.
- Optimization:
  - The IR generated in the frontend is then optimized for either speed or space.
  - Such as loop optimization, dead code elimination etc.
- Back End:
  - It translates the optimized IR into machine code or assembly language specific to the target device.

What is an interpreter?

- It takes the source code, converts it to machine instructions and directly runs them.
- The working of an interpreter is closely associated to how a debugger works.
- This was developed to run projects or apps which are still under construction.

- An interpreter generally uses these following strategies for program execution:
  1. Parse the source code and perform its behavior directly. Early versions of LISP used this.
  2. Translate source code into some efficient intermediate representation and immediately execute that. Perl, Python, MATLAB, and Ruby are some examples.
  3. Explicitly execute stored precompiled bytecode made by a compiler, which is compatible with the interpreter's virtual machine.
- Some systems, such as contemporary versions of BASIC and Java may also combine 2 and 3 types.

Compilers vs Interpreters

- Fun fact: You can write an interpreter for C and a compiler for JavaScript.
- Compiled code is very fast at runtime. Whereas interpreted code takes longer since it translates everything each time u run it.
- Hence, compiled code is beneficial if there is a large codebase, which would take very long time to compile.

- Whereas an interpreter is beneficial in cases where, we need to debug the app, while its still in production.
- Another good feature of interpreted code is that, its very portable, any machine with the interpreter installed can run it.
- The compiler and the interpreter both have their own pros and cons; it depends on us to choose which works best.

Is JIT compiler better?

- The main idea behind a JIT compiler is to compile code during program execution, allowing for optimizations based on program's behavior and the runtime environment.
- A drawback to this is that you have to let the program run for some time to achieve optimized code, meaning apps built using this have longer start-up times.
- A few examples are Java virtual machine, Google's V8 used in chrome.

A deeper look at compilers

- Due to resource limitations, compilers were split up into multiple phases.
- The ability to compile in a single pass has been seen as a benefit because it simplifies the job of writing a compiler.
- Let us look at a three-stage compiler i.e. a compiler which has a frontend, middle end, and a backend.

Front End

- While the frontend can be written as a single function, its traditionally implemented and analyzed as several phases.
- This method enables modularity.
- Most commonly its divided into 3 phases: (assuming preprocessing is already done)
  - Lexical analysis
  - Syntax analysis
  - Semantic analysis

Lexical Analysis

- Also called tokenizer, this breaks down the source code into a sequence of small pieces called lexical tokens. A token is a name, value pair.
- There are 2 phases here:
  - Scanning: segments the text into units called lexemes and assigns them a category.
  - Evaluating: converts lexemes into a processed value.
- Some common token categories (assigned by the scanning phase) may include identifiers, keywords, separators, operators, literals, comments etc.

Syntax Analysis

- Also called parsing, involves going through the token sequence identify the structure of the program
- Using this it then builds a parse tree. Which replaces the linear token sequence with a tree data structure built according to some rules which defines the language’s syntax.

Semantic Analysis

- This takes the parse tree and builds the symbol table, and performs semantic checks such as:
  - Type checking: checking for type errors.
  - Object binding: associating variable and function references with their definitions.
  - Definite assignment: requiring all local variables to be initialized before use.
- In this phase the compiler throws errors if incorrect code is present.

Optimization or Middle End

- There are many methods in which a compiler optimizes the code, here are a few:
- Constant folding: It evaluates the constant expressions. Ex. 2 + 6 is replaced with 8 during compilation.
- Constant propagation: It makes the variables constant if it finds it to be constant throughout. Ex. If x = 10, and y = x + 5; this is replaced with y = 15

- Dead code elimination: It removes that code which does not affect the program's output. Ex. Removing the code after the return statement
- Loop optimization:
  - Loop invariant code motion: brings the code outside the loop if it does not change in the loop.
  - Loop fusion: combines adjacent loops which iterate over the same data.
- Inlining functions: it does this to remove the function call overhead, and allows for further optimizations like constant folding or loop unrolling.

And many more techniques.

Back End

- It takes the intermediate representation, and tries to optimize it using register allocation etc.
- Register Allocation:
  - The compiler decides which variables to store in CPU registers and which in memory.
  - This step is crucial in optimizing the program's performance since accessing registers is faster than memory.
- The final step before the executable is formed is linking. The program may contain many modules, linking combines them into a single executable.

My own compiler:

- The programming language name is “English Syntax Programming New” or ESPN for short.
- It can contain stuff like:

```
can you please print stuff like “Hello World!!!”
```

- Here only print is read by my compiler and it only prints whatever is present in double inverted commas.

- Front End: I’ve made a tokenizer which tokenizes each print keyword, there is no parse tree built since its just one function. It then looks for a string to associate the print keyword to.
- IR (C code): It generates ‘C’ code using the tokens.
- Middle End: There is no optimizer phase for my compiler.
- Back End: Compiles this code using GCC.
- This is essentially a one pass compiler.