Universidad Nacional Autónoma de México
Faculty of Engineering

Computer Engineering.

Compilers Course

COMPILER

STUDENT:
320254536
315293728
423108596
320246881
320250648

Group:
5
Semester:
2026-I

México, CDMX. November 2025

Contents

1 Introduction

In this final report for the Compilers course project, we completed the construction of a full compilation pipeline for a subset of the *Go programming language*. In earlier reports, we implemented a context-free grammar, a lexical analyzer, a LALR(1) parser using RPLY, and a semantic analyzer using a visitor pattern based on Syntax Direct Translation. Building on this, this final phase focuses on generating intermediate code and translating it into an executable program as required. The first new component is the TAC generator, which traverses the abstract syntax tree (AST) and emits a linear intermediate representation using a fixed set of TAC patterns. The second new component is the target code generator, implemented as a source-to-source compiler (transpiler) that translates the validated Go subset into C code. So, for this final stage, the main objective is to implement a complete compile process followed by the next specific objectives:

- To design and implement a Three-Address Code intermediate representatiton for expressions, variable declarations, control-flow structures (if, for) and function calls, using a consistent set of TAC forms that can be generated from the AST.
- To develop a C code generator that maps the AST of the validated Go code, correctly handling control flow and type-inferred declarationes, and producing an executable file for each input source code.
- To integrate the previous stages: lexer, parser and semantic analyzer with TAC generation and the GCC based target code generation into a single pipeline for valid programs.
- To verify the complete program with representative tests cases to validate the generated TAC and the compiled binaries via GCC produce the expected runtime outputs.
- To ensure error handling across the pipeline, such that lexical or syntactic errors stop the process before TAC/C generation, while purely semantic errors are reported as "Parsing Success! SDT error..." and do not produce intermediate or target code.

2 Theoretical Framework

2.1 Intermediate Representation

The front end of a compiler builds and intermediate representation of the source program from which the bach end generates the target program. As we seen in class, there are two kind of intermediate representations [1]:

- Trees, including parse trees and (abstract) syntax trees.
- Linear representations, especially Three-Adress Code.

An Intermediate Representation (IR) is designed to have a simple structure that facilities optimization, analysis, and efficiente code generator [2]. It is possible that a compiler will construct a syntax tree at the same time it constructs steps of thee-adress code [1].

2.2 Abstract Syntax Tree

The abstract syntax tree can be a usable IR if the goal is simply to emit assembly language without much optimization. Then, to generate assembly language, it is as simple as performing a post-order traversal of the AST and emitting a few assembly instructions corresponding to each node [2].

2.3 Three Adress Code

Three-adress code (TAC) will be the intermediate representation as we said, it is a generic assembly language [3] and is a sequence of instructions of the form [1]:

$$x = y \quad \textbf{op} \quad z$$

Where x, y and z are names, constants, or compiler-generated temporaries, and op stands for an operator. This technique is built from two concepts: address and instructions and can be implementing using records with fields for the addresses. The common instruction forms are summarize in the next table [1]:

Instruction type	**General TAC form**	**Description**
Binary assignment	`x = y op z`	Assigns to `x` the result of applying a binary arithmetic or logical operator `op` to `y` and `z`.
Unary assignment	`x = op y`	Assigns to `x` the result of applying a unary operator (unary minus, logical negation, shifts, type conversions, etc.) to `y`.
Copy	`x = y`	Copies the value stored at the address of `y` into the address of `x`.
Unconditional jump	`goto L`	Transfers control to the instruction labeled `L`. The next executed instruction is `L`.
Conditional jump	`if x relop y goto L`	Evaluates the relational operator `relop` (`<`, `==`, `>=`, etc.) between `x` and `y`. If the relation holds, control jumps to `L`; otherwise, the following instruction is executed.

Address and pointer assignment	`x = &y`
	`x = *y`
	`*x = y`
		`x = &y`: `x` receives the address (l-value) of `y`.
		`x = *y`: `x` receives the r-value stored at the location pointed to by `y`.
		`*x = y`: the contents of the location pointed to by `x` are set to the value of `y`.

Table 1: Summary of typical three-address-code instruction forms.

2.4 Intermediate Code Generation (ICG)

The code generator takes as input an intermediate representation of the source program and maps it to the target language [4], following the previously declared forms. The usual workflow would implement an algorithm that generates the intermediate code, which has an instruction for each operator in the tree representation coming from the semantic analyzer [1].

The first techniques involved generating the code for a virtual machine and then expanding it into real machine instructions. Then, new techniques surged, such as generating the code by tree parsing, [4]. The code generator parses the input subject tree, and on each reduction, outputs target code.

2.5 Target Code Generation (TCG)

The final phase of a compiler is the code generator. It receives an intermediate representation (IR) of the source code with additional information from the symbol table to produce the target code, usually in the form of machine code. [5] [6]

Code generator main tasks:

- Instruction selection
- Register allocation and assignment
- Instruction ordering

When generating machine code, high level constructs from the IR are transformed into sequences of low level instructions for the hardware, memory must be allocated for variables and data structures, deciding which variables should be stored in registers or memory as well as the scope and lifetime of each one. Additionally, some code optimization may be performed at this stage, such as constant folding of peephole optimization. [6]

2.6 Linker phase

The linker combines all of the object files produced in the compilation process and the necessary libraries into an executable file [7]. The linker may also add any additional code that is necessary to start or end the program, such as the passing of command line arguments [8]. The linker's main tasks are:

- Symbol resolution: Ensures each symbol pints to the correct definition.
- Relocation: Adjusts addresses to match the final memory layout of the program.
- Optimization: Performs additional optimization.
- Library management: Links the required functions from libraries.
- Debugging information: Adds debugging information to the executable program [9].

Linking can be done in one of two ways:

- Static Linking: All libraries are copied into the executable file.
- Dynamic Linking: The executable code still contains some undefined symbols (usually used for often used libraries) that must be resolved during runtime [8]

2.7 Assembler

The assembly language code, which contains instructions that are translated into processor instructions [10], must be processed by a program in order to generate the machine language code. An assembler is the program that translates the assembly language code into the machine language [11], and produce object files [10].

2.7.1 NASM

NASM (Netwide Assembler), is an open source 80x86 and x86-64 assembler designed for portability and modularity. It supports a large range of object file formats and its syntax is designed to be simple and easy to understand. It supports all currently known x86 architectural extensions [12].

2.8 Transpilers

A transpiler is a type of compiler which reads source code written in high level language and produces the equivalent code in another high level language [13].

3 Development

3.1 Project structure and general functionality

The project is written in python and has the final following structure.

```
unam
README.MD
fi
compilers
g5
05
build
example.c
example.exe
example.log
example.txt
doc
05-Compilers-Final.pdf
05-Compilers-Parser.pdf
05-Compilers-Presentation.pdf
Team05_Compilers_Lexer.pdf
src
main.py
modules
lexer.py
parser.py
savePrint.py
semantic.py
test
fail.go
fib.go
helloworld.go
operation.go
syntaxerr.go
type_mismatch.go
undeclared.go
README.MD
```

3.2 Grammar Specification

The grammar implemented was done by following Go language's own grammar specification [14], but not entirely. The general structure it represents is the following:

```
SourceFile
Package Declaration
Import Declarations
Top-level Declarations
Type Declarations
Function Declarations
Variable Declarations
```

3.2.1 Grammar Notes

The Go language's grammar has the semicolon ";" as a terminal symbol; however, while doing the implementation this proved to introduce ambiguity to the grammar and raise shift/reduce and

reduce/reduce warnings from the parser generator. For which we decided to not include this optional feature and not handle the semicolon as a terminal symbol.

For the syntax, the format used for better understanding inside the documentation is described as follows:

- Each production uses the format: `production_name = Expression`
- ϵ represents an empty production.
- Basic Operators:
  - | for alternatives.
  - [] for optional elements.
  - { } for repetition (zero or more).
  - ( ) for grouping.
- Terminal symbols are shown in **bold** for keywords, quoted for operators/punctuation, or token names in uppercase (IDENT, LIT_STR, etc.).

In general, the grammar implements supports a Go-like syntax with a simplified type system.

3.3 Intermediate Code Generation

Following the semantic analysis phase, the compiler generates an intermediate representation of the source program using Three-Address Code (TAC). This intermediate form serves as a bridge between the high-level abstract syntax tree (AST) and the final target code, enabling multiple optimizations and facilitating portability across different architectures. This phase marks the beginning of the synthesis stage in the compilation process, where the compiler transitions from analyzing the source program to constructing its executable representation.

It is important to note two parallel solutions were implemented. The first one uses the TAC intermediate representation and transforms it into 64-bit NASM assembly, then leverages the GCC toolchain to produce an executable file, skipping the optimization phase, but remaining extensible for more advanced transformations later on. While the second one, uses C as its intermediate code and with a blackbox approach GCC is in charge of the translation to machine code, and then linking.

Traversal Strategy (Visitor Pattern)

The same traversal strategy used in the semantic analyzer is applied here: the *Visitor* pattern, implemented by the `TacGenerator` class. The generator traverses the AST, and for each type of node there is a corresponding visit method. Temporary variables are created through the method `new_temp()`, which produces a unique temporary name using an internal counter.

We use the Three-Address Code forms shown in Table 2.

Control Flow and Function Patterns

The TAC generator follows consistent structural patterns for representing control flow constructs such as `if` statements, `for` loops, and function definitions. These patterns are shown below.

For Loop Pattern

```
init_statements
GOTO condition_label
body_label:
    loop_body
```

No.	TAC Form	Usage in the Provided Code
1	`x = constant`	Direct constant assignment.
2	`GOTO L`	Unconditional jump to label L.
3	`LABEL L:`	Label definition for control flow targets.
4	`x = y`	Simple assignment between variables.
5	`t = a op b`	Binary operation with assignment to a temporary.
6	`x = t`	Assignment of temporary result to a variable.
7	`x = x op constant`	In-place operation with a constant.
8	`IF_TRUE t GOTO L`	Conditional jump if the temporary is true.
9	`RETURN x`	Return value from a function.
10	`END_FUNC name`	End of function definition.
11	`FUNC name:`	Start of function definition.
12	`t = CALL func arg`	Function call with argument, storing result.
13	`t = CALL t`	Function call using a temporary.
14	`CALL func arg`	Function call without storing a result.

Table 2: Three-Address Code Forms Used in the Program

```
condition_label:
    t = condition
    IF_TRUE t GOTO body_label
next_label:
```

If Statement Pattern

```
t = condition
IF_TRUE t GOTO label
[else_block]   # Optional
label:
```

Function Definition Pattern

```
FUNC function_name:
    function_body
    RETURN value
END_FUNC function_name
```

3.4 Target Code Generation

3.4.1 NASM code

What the target code generation module does for the first solution, could be described by the following:

- Maps TAC operators (ADD, MUL) to corresponding instructions (add, imul)
- Converts TAC jumps and labels to assembly branches using jmp, je, jne with NASM label syntax and with two passes.
- Function calls by implementing Windows x64 calling convention.
- Generates appropriate extern declarations for C runtime functions (printf, exit).
- TAC temporaries (t0, t1) to either x86-64 registers or stack locations based on variable liveness and scope
- Implements proper stack frame alignment per Windows x64 calling convention, including:

3.4.2 Transpilation to C

The final stage of our compiler pipeline is the generation of executable machine code. Instead of generating Assembly language directly, we opted for a **source-to-source compilation (transpilation)** strategy.The compiler translates the source Go code into **C language**, which is then passed to the GCC compiler to handle optimization, linking, and binary generation. The implementation is encapsuled in the **CCodeGenetator** class within **codegen.py**.

3.4.3 Traversal Strategy (Visitor Pattern)

Similar to the semantic analyzer, the code generator employs the **Visitor Pattern**. This class iterates over the Abstract Syntax Tree (AST) genetated by the parser. For each node type (e.g., IfStmt, FunctionDecl), a specific method **visit_NodeName** produces the equivalent C syntax.

- **State Management:**The generator maintains an **indent_level** to ensure the output C code is properly indented and readable, facilitating debugging.
- **Buffer:** Code strings are accumulated in a list and finally joined with standard C headers (`#include <stdio.h>`, `#include <stdbool.h>`)

3.4.4 Mapping Go Constructors to C

The translation involves several structural transformations to bridge the gap between Go and C:

1. **I/O Operations (fmt.Println):** Since C uses **printf** which requires strict type formatting, the generator performs a heuristic check on the argument type.
   - If the argument is a string literal, it generates `printf("string\n")`
   - If the argument is an integer expression (like in the Fibonacci example), it generates `printf("%d\n",value)`
2. **Control Flow (For Loops):** Go's for loop syntax allows declaring loop-local variables (e.g. for i := 0;...). In standard C generation, simply declaring **int i = 0;** before a while loop would place the variable **i** in the function's main scope. This creates a semantic error if two sequential loops attempt to declare the same variable **i**, causing a "redefinition" conflict. For this, our generator implements **Scope Isolation**. It wraps the entire loop structure within a dedicated block scope in C. This ensures that variables declared in the loop initialization do not leak into the outer scope, respecting Go's scoping rules.
3. **Variable Declarations:**
   - **Type Inference:** For short declarations, the generator infers the C type. Integer literals map to **int**, floating points to **double**, and strings to **char***.
   - **Multiple Assignament:** Parallel declarations in Go (e.g. var n1, n2 int = 0, 1) are serialized into sequential C declarations (int n2 = 0;, int n1 = 1;).
4. **Entry Point Adaptation:** The compiler handles the semantic differences between Go and C entry points. While Go's **func main()** has no return type, the C standard requires `main` to return an integer. The code generator detects the `main` function identifier and automatically injects an `int` return type and a return code (0) if implied, ensuring compatibility with the GCC linker expectations.
5. **Expressions and Operators:** Since Go and C share similar operator precedence rules for arithmetic and logical operations the translation of `BinaryExpr` and `UnaryExpr` nodes is performed via direct 1-to-1 mapping. Parentheses are preserved from the AST structure to guarantee that the evaluation orfer in the generated C code matches the original Go source intent.

We consider that integrating GCC as backend provided a robust solution for the final compilation stage. This approach allowed us to successfully transpile complex Go constructs, such as recursion and control flow, into valid C code.

3.5 Linking

The system internaly runs the following command, to perform the assembler processing, and convert assembly language into machine-readable object code.

```
nasm -f win64 input.asm -o output.obj
```

Then for the final phase, linking to GCC it performs the command in order to transform the object code into a standalone Windows executable.

```
gcc input.obj -o output.exe
```

3.6 Usage considerations

The program is built around having a single file input, it does not support single line instructions, or otherwise. There are two modes of execution given by the terminal option `-f`. Example of usage:

```
$ python main.py sourceFile.go --f
$ python main.py sourceFile.go
```

The 'f' option generates intermediate files for transparency, which include:

- A .txt file that includes the token summary output by the lexer, the parse tree generated by the parser, and the TAC intermediate code in on.
- A .c file

4 Results

In the following results we will just report images of the new implementations: TAC and Target Code Generation. The previous stages will be only described briefly.

4.1 Test Cases

4.1.1 Hello, World!

OP01	**Code test for `helloworld.go` with TAC and C backend**: Lexer *Valid* Syntax *Valid* Semantic *Valid* TAC generation *Generated* Target C code *Generated*
Go program valid that prints `"Hello, World!"` and is now translated to three-address code (TAC), then to C, compiled with `gcc` and executed.	
Steps to follow to perform the test	**Expected Result**
• Run the full compilation pipeline on `helloworld.go` • Internally, follows the stages: lexical analysis, parsing and SDT-based semantic analysis over the parse tree. • If semantic analysis succeeds, invoke the TAC generator on the parse tree. • Use the SDT as input and call `gcc` on the generated C file to build the executable `helloworld.exe` • Execute the compiled binary and inspect the console output.	• *Analysis Stage*: Same as previous tests: valid token stream, `Parsing Success!`, and parse tree generates. • *TAC phase*: console prints `TAC generated`; a linear three-address code representation is constructed and save in `...\build`. • `C code generated`: console prints `"Generating C code..."` and a valid `helloworld.exe` is produced. • *Execution*: console shows `-- EXECUTING helloworld.exe --` followed by `Program output:` and the line `Hello, World!`.

```
TAC generated

Generating C code...
C code generated

Compiling with GCC -> helloworld.exe...
Successful Compilation!

--- EXECUTING helloworld.exe ---

Program output:
Hello, World!


--- EXIT CODE 0 ---
```

Figure 1: Console output for helloworld.go

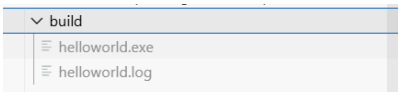

Figure 2: Files generated by the input helloworld.go

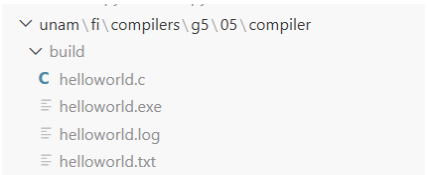

Figure 3: Files generated using -f

4.1.2 Fibonacci sequence

OP02	**Code test for `fib.go` with TAC and C backend:** Lexer *Valid* Syntax *Valid* Semantic *Valid* TAC generation *Generated* Target C code *Generated*
Go program that computes the n-th Fibonacci number iteratively and is now translated to three-address code (TAC), then to C, compiled with `gcc` into `fib.exe` and executed.	
Steps to follow to perform the test	**Expected Result**
• Run the full compilation pipeline on `fib.go`. • Internally, follow the stages: lexical analysis, parser and semantic analysis. • If semantic analysis succeeds, uses the STD as input for the C backend to emit `fib.c` and call `gcc` to build the executable `fib.exe`. • Execute fib.exe and inspect the console output.	• *Analysis Stage*: Same as previous tests: valid token stream, `Parsing Success!`, and parse tree generated. • *TAC phase*: console prints `Starting TAC Generation...`, `TAC Generation Success!` and `TAC generated`; a three-address code representation is constructed and saved in `...\build`. • *C backend*: console prints `Generating C code...` and `C code generated`; then `Compiling with GCC -> fib.exe...` followed by `Successful Compilation!`. • *Execution*: console shows `-- EXECUTING fib.exe --` followed by `Program output:` and the line `34`.

```
Starting TAC Generation...
TAC Generation Success!
TAC generated

Generating C code...
C code generated

Compiling with GCC -> fib.exe...
Successful Compilation!

--- EXECUTING fib.exe ---

Program output:
34


--- EXIT CODE 0 ---
```

Figure 4: Console output for fib.go

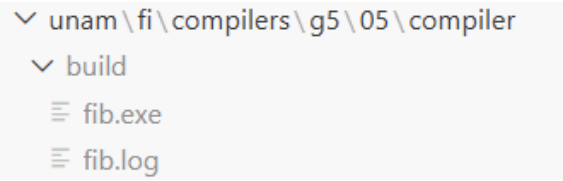

Figure 5: Files generated by the program when the input is fib.go

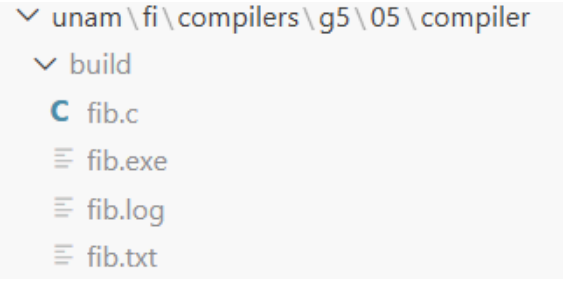

Figure 6: Files generated using -f

4.1.3 Operation

OP03	**Code test for `operation.go` with TAC and C backend:** Lexer *Valid* Syntax *Valid* Semantic *Valid* TAC *Generated* Target Code *Generated*
Go program that performs a simple arithmetic assignment `a := 2 + 2` in `main()`, which is translated to three-address code (TAC), then to C, compiled with `gcc` into `operation.exe` and executed.	
Steps to follow to perform the test	**Expected Result**
• Run the full compilation pipeline on `operation.go`. • Follow the previous stages: lexical analysis, parser and semantic analysis (SDT) over the parse tree. • If semantic analysis succeeds, use the SDT structure as input for the C backend to emit `operation.c` and call `gcc` to build the executable `operation.exe`. • Execute `operation.exe` and inspect the console output.	• *Analysis Stage*: Valid token stream, `Parsing Success!`, and parse tree generated; console prints `Starting Semantic Analysis (SDT)...` and `SDT Verified!`. • *TAC phase*: console prints `Starting TAC Generation...`, `TAC Generation Success!` and `TAC generated`; a three-address code representation of `a := 2 + 2` is constructed and saved in `...\build`. • *C backend*: console prints `Generating C code...` and `C code generated`; then `Compiling with GCC -> operation.exe...` followed by `Successful Compilation!`. • *Execution*: console shows `-- EXECUTING operation.exe --` followed by `Program output:` and the line `4`.

```
Starting TAC Generation...
TAC Generation Success!
TAC generated

Generating C code...
C code generated

Compiling with GCC -> operation.exe...
Successful Compilation!

--- EXECUTING operation.exe ---

Program output:
4
```

Figure 7: Console output for operation.go

Figure 8: Files generated by the input operation.go

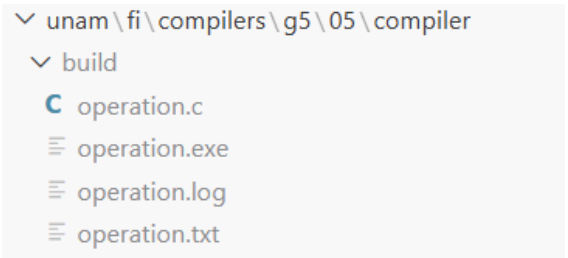

Figure 9: Files generated with –f

ID	Program / Scenario	Lexer	Syntax	Semantic / Main error
OP04	Expression `x + *y` with commented blocks.	Valid	Invalid	SDT not executed. Parser reports `Parsing error: Unexpected token '*'` at the position of `*`.
OP05	`x` declared as `int` and later assigned `"hola"`.	Valid	Valid	Invalid. SDT type-checking reports assignment incompatibility: `int` variable receives a `string`.
OP06	Assignment `x = 10` in `main()` without prior declaration.	Valid	Valid	Invalid. SDT reports `undeclared identifier 'x' in current scope`; SymbolTable for `main` has no entry for `x`.
OP07	Program using invalid operator character `@` in an expression.	Invalid	Not executed	Not executed. Lexer stops after detecting the first invalid token `@`; comments and whitespace before it are ignored.

Table 3: Negative test programs and where errors are detected in the compilation pipeline.

5 Conclusions

This project successfully follows the complete process of a compiler, putting into practice what was learned during the course. It successfully evolved from each stage of analysis to the target code. Together, these stages allow the tool to take Go source code, validate it lexically, syntactically and semantically, then produce an intermediate TAC representation, generate equivalent C code, and finally obtain an executable via GCC. Overall, the implementation meets the course goal of building a working compiler prototype, while keeping the architecture modular to future extensions such as additional language features or optimization over TAC.

Regarding the objectives defined at the beginning, we can affirm that they were effectively achieved throughout this final deliverable. Three Address Code representation was successfully carried out through a visitor-based implementation that traverses the AST and generates the instructions for each source code. In parallel, the development of a C transpiler was completed, because it helped: the validated Go source code is translated into readable and executable C code, which preserves the originals program's structure. Finally, the integration of all components into a single pipeline was verified using both positive and negative test cases. For valid programs, the compiler consistently progresses from tokens to parse tree, semantic validation, TAC, C source code, and a working executable whose output matches expectations. For invalid programs, the system correctly stops at the appropriate stage without generating the next phases. So, this confirms that not only we achieved the specific objectives, but also that the resulting project behaves as a compiler for the chosen subset of Go.

References

[1] R. S. A. A.V. Abo M.S. Lam and J. D. Ullman, *Compilers : principles, techniques, and tools*. Pearson Education. Addison-Wesley-, 2007, ISBN: 0-321-48681-1.

[2] D. Thain, *Introduction to Compilers and Language Design*, 2nd ed. Self-published, 2023, Revision Date: August 24, 2023, ISBN: 979-8-655-18026-0. [Online]. Available: `http://compilerbook.org`.

[3] M. Johnson and J. Zelenski, *Three address code examples*, CS143 Handout 24, Stanford University, Course handout, 2012. Accessed: Dec. 1, 2025. [Online]. Available: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/240%20TAC%20Examples.pdf.

[4] Salgado Cruz Emiliano Roman, *Notas del curso de compiladores*, UNAM, Facultad de Ingeniería, Notas personales con base en las clases., 2025.

[5] V. Arora, "Chapter 8 Code Generation," en, [Online]. Available: https://pg.its.edu.in/sites/default/files/Code%20Generation.pdf.

[6] *Lesson 8: Code Generation*, en. [Online]. Available: https://btu.edu.ge/wp-content/uploads/2023/08/Lesson-8_-Code-Generation.pdf.

[7] *0.5 — Introduction to the compiler, linker, and libraries – Learn C++*. Accessed: Dec. 2, 2025. [Online]. Available: https://www.learncpp.com/cpp-tutorial/introduction-to-the-compiler-linker-and-libraries/.

[8] *Compiling a C Program: Behind the Scenes*, en-US, Section: C Language. Accessed: Dec. 2, 2025. [Online]. Available: https://www.geeksforgeeks.org/c/compiling-a-c-program-behind-the-scenes/.

[9] *Linker*, en-US, Section: Software Engineering. Accessed: Dec. 2, 2025. [Online]. Available: https://www.geeksforgeeks.org/software-engineering/linker/.

[10] R. Hyde, *LINUX Assembly Language Programming*. Prentice Hall PTR, 2000, ISBN: 9780130879400.

[11] S. P. Dandamudi, *Guide to Assembly Language Programming in Linux*. Springer, 2005, ISBN: 978-0387-26171-3.

[12] The NASM Development Team, *Nasm – the netwide assembler version 3.01*, https://www.nasm.us/xdoc/3.01/nasmdoc.pdf, Accessed: December 1, 2025.

[13] Sengstacke, P., *Javascript transpilers: What they are and why we need them*, https://www.digitalocean.com/community/tutorials/javascript-transpilers-what-they-are-why-we-need-them, Accessed: December 1, 2025.

[14] The Go Authors. "The go programming language specification." version go1.25, Accessed: Nov. 3, 2025. [Online]. Available: https://go.dev/ref/spec.