



# Compiler - Final Project

Facultad de Ingeniería

Compilers

Semester 2025-2

Group: 5

**Team 1 Teammates:**

Calderón Olalde Enrique Job

Salazar Islas Luis Daniel

Tepal Briseño Hansel Yael

Ugartechea González Luis Antonio

**Due Date:** 08/06/2025

# 1 Introduction

## Problem Statement

We were asked to create a compiler for a programming language, this compiler needed to be able to read a source code file, filter it through the different stages of compilation. The compiler should be able to do at least if statements and arithmetic operations.

## Motivation

We were motivated to work with a C based programming language, this because it is easier to read and write. Nevertheless, we need to keep in mind that these languages need to be translated to machine code, this is also a benefit from the C base. Thanks to the structure, we are able to make the bridge between human-readable code and the machine-executable instructions.

## Objectives

Some of the objectives of the general assignment are:

- create a string of tokens that can be used to represent the source code.
- produce a parse tree that represents the structure of the source code.
- produce an abstract syntax tree that represents the structure of the source code.
- produce an intermediate representation of the source code.
- produce an executable file.
- the executable file should be able to run the code written in the source code file.

# 2 Theoretical Framework

## Compiler Architecture and Design

A compiler is a complex software system that translates programs written in a high-level programming language into machine code or another lower-level representation. The compilation process is traditionally divided into several distinct phases, each with specific responsibilities and well-defined interfaces. Our implementation follows the classical multi-phase compiler architecture, consisting of five main components that work sequentially to transform source code into executable assembly code.

### Multi-Phase Compilation Pipeline

The compilation process in our implementation follows a linear pipeline architecture where each phase processes the output of the previous phase:

1. **Lexical Analysis Phase:** transforms raw source code into a stream of tokens
2. **Syntax Analysis Phase:** converts tokens into an abstract syntax tree (AST)
3. **Semantic Analysis Phase:** validates semantic correctness and builds symbol tables

4. **Intermediate Code Generation Phase:** transforms AST into intermediate representation (IR)
5. **Code Generation Phase:** converts IR into target machine assembly code

This modular approach provides several advantages including separation of concerns, easier debugging and testing, and the ability to make small changes to the compiler to adapt new features.

## Lexical Analysis Theory

### Regular Languages

Lexical analysis is based on the theory of regular languages. Each token type in our language is defined by a regular expression that describes the pattern of characters that form valid instances of that token type. The lexical analyzer uses these patterns to scan the input character stream and identify tokens.

Our implementation uses Python's `re` module, which implements a regular expression engine. The token specification includes:

- **Keywords:** fixed strings like `if`, `else`, `while`, `int`, `return`, `print`
- **Identifiers:** pattern `[a-zA-z_][a-zA-z0-9_]*` for variable and function names
- **Constants:** pattern `-?[0-9]+` for integer literals
- **Operators:** multi-character operators such as `==`, `!=`, `<=`, `>=`, `&&`, `||`
- **Literals:** string patterns enclosed in quotes with escape sequence support
- **Punctuation:** single-character delimiters and brackets

### Tokenization Process

The tokenization process uses a longest-match principle implemented through a combined regular expression. All token patterns are concatenated using the alternation operator (`|`) with named capture groups, allowing the lexer to identify both the token type and value in a single pass through the input.

## Syntax Analysis Theory

### Context-Free Grammars

Syntax analysis is based on our context-free grammar (CFG). Our language is defined by a context-free grammar that specifies the syntactic structure of valid programs. The grammar uses a notation to define production rules that describe how complex language constructs are formed from simpler components.

The grammar for our C-based language includes productions for:

- program structure
- statement types
- control flow constructs

- function definitions and calls

## Recursive Descent Parsing

Our parser implementation uses the recursive descent parsing technique, which is a top-down parsing method that constructs the parse tree from the root down to the leaves. Each non-terminal in the grammar corresponds to a parsing method that recognizes that syntactic construct.

The recursive descent approach offers several advantages:

- direct correspondence between grammar rules and parsing methods
- easy to implement and understand
- good error recovery capabilities

## Abstract Syntax Trees

The parser constructs an Abstract Syntax Tree (AST) as its primary output. The AST is a tree representation of the syntactic structure of the program where:

- each node represents a language construct (statement, expression, declaration)
- leaf nodes represent terminals (identifiers, constants, operators)
- internal nodes represent syntactic categories and their relationships
- irrelevant syntactic details (like parentheses and keywords) are abstracted away

## Semantic Analysis Theory

### Symbol Tables and Scope Management

Semantic analysis uses symbol table data structures to track identifier declarations and their properties throughout the compilation process. Our implementation uses a stack-based symbol table that supports nested scopes:

- **Function Scope:** contains local variable declarations within functions
- **Block Scope:** supports nested scopes within control structures

Each symbol table entry contains:

- identifier name
- type
- scope level

### Type Checking and Semantic Validation

The semantic analyzer performs several types of validation:

1. **Declaration Checking:** ensures all identifiers are declared before use
2. **Type Checking:** validates type compatibility in expressions and assignments

3. **Scope Resolution:** ensures identifiers are used within their valid scope

## Visitor Pattern Implementation

The semantic analyzer uses the Visitor design pattern to traverse the AST.

## Intermediate Code Generation Theory

### Three-Address Code

Our intermediate representation is based on three-address code (TAC), a form of intermediate code where each instruction has at most three operands. This representation has several advantages:

- explicit representation of intermediate values using temporary variables
- easy translation to assembly language
- suitable for optimization passes
- machine-independent representation

### IR Instruction Set

The intermediate representation includes the following instruction types:

- **Assignment Instructions:** `x = y`, `x = 5`
- **Binary Operation Instructions:** `t1 = x + y`, `t2 = a == b`
- **Label Instructions:** `l2:`, `main:`
- **Jump Instructions:** `goto l2`, `if_false t1 goto l3`
- **Function Instructions:** `call func`, `return x`
- **I/O Instructions:** `print x`, `print "hello"`

### Control Flow Translation

The IR generator translates high-level control structures into low-level jump instructions:

- **Conditional Statements:** translated using conditional jumps with labels for if-else branches
- **Loop Constructs:** implemented using label-jump combinations for loop entry, condition checking, and exit
- **Function Calls:** represented as call instructions with proper label management

## Code Generation Theory

### Target Architecture: x86-32

Our code generator targets the x86-32 architecture running on Linux. Key architectural features utilized include:

- **Register Set:** `eax`, `ebx`, `ecx`, `edx` for general computation; `esp`, `ebp` for stack management

- **Memory Model:** 32-bit linear addressing
- **Calling Convention:** standard x86 calling convention with stack-based parameter passing

## Stack Frame Management

The code generator implements standard stack frame management using the EBP (base pointer) register:

- **Function Prologue:** save old ebp, set new ebp, allocate local variable space
- **Variable Access:** use ebp-relative addressing for local variables
- **Function Epilogue:** deallocate locals, restore ebp, return to caller

## System Call Interface

The generated code uses Linux system calls for I/O operations:

- **sys\_write (4):** for output operations (print statements)
- **sys\_exit (1):** for program termination
- **interrupt 0x80:** software interrupt mechanism for system call invocation

## Runtime Support Routines

The code generator includes runtime support routines for operations not directly supported by the instruction set:

- **Integer to String Conversion:** implements division-based algorithm for printing integers
- **String Output:** handles utf-8 encoded string literals with escape sequence processing
- **Memory Management:** manages static data section for string literals and buffers

## Language Design Constraints

Our simplified C-based language has specific design constraints that influenced the theoretical approach:

- **Single Data Type:** only 32-bit signed integers are supported, simplifying type checking and code generation
- **No Function Parameters:** functions cannot accept parameters, eliminating parameter passing complexity
- **No Dynamic Memory:** no pointers or arrays, resulting in purely stack-based memory management
- **Limited Control Flow:** no for loops or goto statements, using only if-else and while constructs

These constraints allow for a cleaner implementation while still demonstrating all major compiler construction principles.

## Error Handling and Recovery

### Error Detection Strategies

Each compilation phase implements specific error detection mechanisms:

- **Lexical Errors:** invalid character sequences detected during tokenization
- **Syntax Errors:** grammar rule violations detected during parsing
- **Semantic Errors:** type mismatches and scope violations detected during semantic analysis
- **Code Generation Errors:** invalid IR instructions detected during assembly generation

### Error Reporting

The compiler provides detailed error messages including:

- error type and description
- source location information
- context information to help identify the problem

## 3 Development

The compiler is developed in Python, we chose this language for its object-oriented capabilities to implement the different phases of compilation in a modular and understandable manner. Each phase is encapsulated in its own Python module facilitating independent development and testing.

### Lexical Analyzer Implementation

The lexical analyzer (lexer) is implemented using Python's built-in `re` module for regular expression matching. The core of the lexer defines a list of token specifications, where each specification is a tuple containing a token type (e.g., `KEYWORD`, `IDENTIFIER`, `OPERATOR`) and a regular expression pattern that defines the lexeme for that token. These individual regular expressions are combined into a single master regex using the OR operator (`|`), with named capture groups corresponding to each token type. This allows the `re.finditer()` method to efficiently scan the input source code and yield match objects. For each match, the lexer determines the token type from the matched group name and extracts the token value. Comments and whitespace are identified and discarded. String literals are processed to remove their surrounding quotes. This approach directly implements the theoretical concept of using finite automata (as realized by the regex engine) to recognize regular language patterns in the input stream.

### Parser Implementation

The syntax analyzer (parser) is implemented as a recursive descent parser. This top-down parsing strategy is realized by a set of mutually recursive Python methods, where each method typically corresponds to a non-terminal symbol in the language's context-free grammar. Abstract Syntax Tree (AST) nodes are defined as Python classes (e.g., `ProgramNode`, `FunctionNode`, `BinaryOpNode`), each encapsulating the relevant information for a specific language construct. The parser consumes

the stream of tokens produced by the lexer. As parsing methods are called, they check the current token against expected token types or values based on the grammar rules. If a rule is matched, the method consumes the corresponding tokens and recursively calls other parsing methods to process sub-constructs, building up AST nodes in the process. For instance, a method for parsing a binary expression would expect an operand, then an operator, then another operand, recursively calling an expression parsing method for the operands. Error handling is incorporated by raising custom exceptions when the token stream does not conform to the expected syntax, providing messages about the nature and location of the error.

## Semantic Analyzer Implementation

The semantic analyzer is implemented to traverse the AST produced by the parser, performing validation checks and building symbol tables. The Symbol Table is implemented as a Python class, typically using a list of dictionaries to represent the stack of scopes. Each dictionary maps identifier names to **Symbol** objects, which store information like the identifier's type and scope level. Methods are provided to **enter\_scope**, **exit\_scope**, **declare** a new symbol, and **lookup** an existing symbol. The AST traversal is achieved using the Visitor design pattern. A **SemanticAnalyzer** class contains **visit\_NodeType** methods for each type of AST node. When the **analyze** method is called with the root of the AST, it dispatches to the appropriate visit method based on the node's type. During traversal, these visit methods implement semantic rules:

- For **DeclarationNode**, an identifier is added to the current scope in the symbol table.
- For **IdentifierNode**, a lookup is performed to ensure the identifier has been declared.
- For **AssignmentNode** and **BinaryOpNode**, type checking is performed to ensure compatibility between operands (though in our simplified language, this is mainly about ensuring variables are used in contexts appropriate for integers because the language just contains int)
- Function declarations and calls are checked for consistency.

Semantic errors, such as using an undeclared variable or re-declaring a variable in the same scope, result in custom **SemanticError** exceptions being raised.

## Intermediate Code Generator Implementation

The Intermediate Representation (IR) generator also employs the Visitor pattern to traverse the AST. IR instructions (like **LabelInstr**, **AssignInstr**, **BinaryOpInstr**, **JumpInstr**) are defined as Python classes. Each class stores the components of a three-address-like instruction (e.g., target, source1, operator, source2). The **IRGenerator** class maintains a list to which new IR instruction objects are appended. It also manages the creation of unique temporary variable names and labels needed for control flow. As the visitor methods traverse the AST:

- Expressions (e.g., **BinaryOpNode**) are translated into a sequence of IR instructions that compute the expression's value, storing the result in a new temporary variable. The name of this temporary variable is then returned to be used by parent AST nodes.
- Assignment statements generate **AssignInstr** objects.
- Control flow constructs like **ConditionalNode** (if statements) and **WhileNode** (while loops) generate **LabelInstr** for targets and **ConditionalJumpInstr** or **JumpInstr** to manage the



flow of execution. New labels are generated as needed.

- Function definitions generate entry labels and return instructions.

The result of this phase is a flat list of Python objects, each representing an IR instruction in sequential order.

## Code Generator Implementation

The final code generator takes the list of IR instruction objects and translates them into x86 assembly language for a Linux target. The `CodeGenerator` class in Python maintains separate lists for the `.data` and `.text` sections of the assembly file. It iterates through the IR instructions:

- A key aspect is managing variable locations. For each function, it first collects all local variables and temporary variables used. Based on this, it calculates the required stack space for the function's stack frame. Variables are then mapped to offsets from the base pointer (`ebp`).
- `LabelInstr` from the IR translates directly to assembly labels.
- `AssignInstr` translates to `mov` instructions, moving data between registers, memory locations (stack variables), and immediate values.
- `BinaryOpInstr` translates to corresponding x86 arithmetic or comparison instructions (e.g., `add`, `sub`, `imul`, `idiv`, `cmp` followed by `sete`, `setne`, etc.). Operands are loaded into registers (typically `eax`, `ebx`), the operation is performed, and the result is stored back into the target variable's stack location.
- `JumpInstr` and `ConditionalJumpInstr` translate to `jmp`, `je`, `jne`, etc., using the labels defined in the IR.
- `PrintInstr` is handled by generating calls to helper assembly routines for printing integers or by directly using Linux `sys_write` system calls for string literals. String literals are added to the `.data` section with unique labels.
- Function prologues (setting up `ebp`, allocating stack space) and epilogues (restoring `ebp`, deallocating stack space, `ret`) are generated for each function. The `main` function's return is translated into a program exit sequence using the `sys_exit` system call.

Helper routines, such as `print_integer` (which converts an integer in a register to its ASCII string representation and prints it using `sys_write`) and `print_newline`, are appended to the `.text` section. The final assembly code is produced by concatenating the `.data` and `.text` sections.

This modular Python implementation allows each theoretical concept of compilation to be clearly mapped to a corresponding part of the codebase, making the compiler's structure and operation transparent and maintainable.

## Test

To ensure the correctness and robustness of each component of the compiler (Lexer, Parser, Semantic Analyzer, Intermediate Code Generator, and Code Generator), a comprehensive suite of

unit tests was developed using Python's built-in `unittest` framework. These tests cover a wide range of scenarios, including valid inputs, edge cases, and error conditions for each phase.

The tests are organized within the `tests/` directory, with separate files for each compiler module (e.g., `test_lexer.py`, `test_parser.py`).

## Running the Tests

All tests can be executed from the root directory of the project using the following command:

```
1 python -m unittest discover -s tests
```

This command instructs the `unittest` module to:

- `discover`: Automatically find test files.
- `-s tests`: Start discovery in the `tests` directory.

The output of the test suite, indicating the number of tests run and their status (e.g., OK, FAILED, ERROR), is typically printed to the standard output and standard error. To capture this output for review, it can be redirected to a file. For example, to save the results to `tests/test_result.txt`:

```
1 python -m unittest discover -s tests > tests/test_result.txt 2>&1
```

## Expected Test Output

A successful run of all tests will produce output similar to the following in `tests/test_result.txt`:

```
1 .....
2 -----
3 Ran 94 tests in 0.015s
4
5 OK
```

This output indicates that all 94 tests passed successfully. The series of dots (.....) represents individual test cases passing.

## 4 Results

We achieved our expected goals, the compiler is able to receive a source code file and then output an executable file which we can run in our personal computers. We design many test cases to test the compiler, these test cases are designed to cover all the features of the language, including arithmetic operations, if statements, while loops, and function calls. The compiler is able to handle all of these test cases without any issues.

## Example

Here is an example of a source code file that can be compiled with our compiler:



```
int hello() {  
    print("Hello!\n");  
    return 0;  
}  
  
int main(){  
    int i = 0;  
    while (i < 10) {  
        hello();  
        if (i == 5) {  
            print("Halfway there!\n");  
        } else {  
            print(i);  
        }  
        i = i + 1;  
    }  
    print("Goodbye!\n");  
    return 0;  
}
```

Figure 1: Example source code

```

Tokens:
('keyword', 'int')
('identifier', 'hello')
('punctuation', '(')
('punctuation', ')')
('punctuation', '{')
('keyword', 'print')
('punctuation', '(')
('literal', 'Hello!\n')
('punctuation', ')')
('punctuation', ';')
('keyword', 'return')
('constant', '0')
('punctuation', ';')
('punctuation', '}')
('keyword', 'int')
('identifier', 'main')
('punctuation', '(')
('punctuation', ')')
('punctuation', '{')
('keyword', 'int')
('identifier', 'i')
('operator', '=')
('constant', '0')
('punctuation', ';')
('keyword', 'while')
('punctuation', '(')
('identifier', 'i')
('operator', '<')
('constant', '10')
('punctuation', ')')
('punctuation', '{')
('identifier', 'hello')
('punctuation', '(')
('punctuation', ')')
('punctuation', ';')
('keyword', 'if')
('punctuation', '(')
('identifier', 'i')
('operator', '==')
('constant', '5')
('punctuation', ')')
('punctuation', '{')
('keyword', 'print')
('punctuation', '(')
('literal', 'Halfway there!\n')
('punctuation', ')')
('punctuation', ';')
('punctuation', '}')
('keyword', 'else')
('punctuation', '{')
('keyword', 'print')
('punctuation', '(')
('identifier', 'i')
('punctuation', ')')
('punctuation', ';')
('punctuation', '}')
('identifier', 'i')
('operator', '=')
('identifier', 'i')
('operator', '+')
('constant', '1')
('punctuation', ';')
('punctuation', '}')
('keyword', 'print')
('punctuation', '(')
('literal', 'Goodbye!\n')
('punctuation', ')')
('punctuation', ';')
('keyword', 'return')
('constant', '0')
('punctuation', ';')
('punctuation', '}')

```

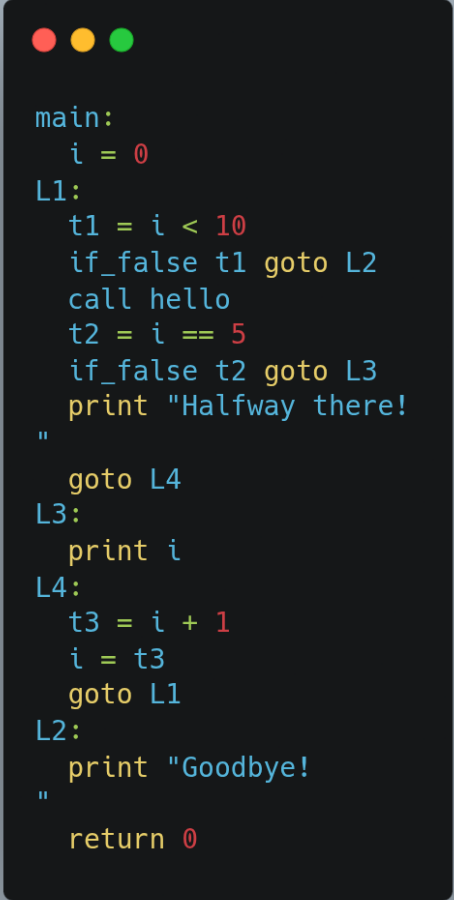
Figure 2: Example tokenization

```

Abstract Syntax Tree (AST):
ProgramNode:
  FunctionNode: hello() -> int
  Block:
    PrintNode:
      LiteralNode: "Hello!\n"
    Return:
      ConstantNode: 0
  FunctionNode: main() -> int
  Block:
    DeclarationNode: i (int)
    Initializer:
      ConstantNode: 0
    WhileNode:
      Condition:
        BinaryOpNode: <
        Left:
          IdentifierNode: i
        Right:
          ConstantNode: 10
      Block:
        FunctionCallNode: name=hello
        ConditionalNode:
          Condition:
            BinaryOpNode: ==
            Left:
              IdentifierNode: i
            Right:
              ConstantNode: 5
          If True:
            PrintNode:
              LiteralNode: "Halfway there!\n"
          Else:
            PrintNode:
              IdentifierNode: i
            AssignmentNode: i =
            BinaryOpNode: +
            Left:
              IdentifierNode: i
            Right:
              ConstantNode: 1
            PrintNode:
              LiteralNode: "Goodbye!\n"
        Return:
          ConstantNode: 0
hello:
  print "Hello!"
"
  return 0

```

Figure 3: Example parse tree



```
main:
  i = 0
L1:
  t1 = i < 10
  if_false t1 goto L2
  call hello
  t2 = i == 5
  if_false t2 goto L3
  print "Halfway there!"
  "
  goto L4
L3:
  print i
L4:
  t3 = i + 1
  i = t3
  goto L1
L2:
  print "Goodbye!"
  "
  return 0
```

Figure 4: Example abstract syntax tree



```
Generated x86 Assembly Code written to example.asm

Program output:
Hello!
0
Hello!
1
Hello!
2
Hello!
3
Hello!
4
Hello!
Halfway there!
Hello!
6
Hello!
7
Hello!
8
Hello!
9
Goodbye!
```

Figure 5: Example output

In this example, we can see how the compiler is able to tokenize <sup>2</sup> the source code <sup>1</sup>, parse it into a parse tree <sup>3</sup>, and then generate an abstract syntax tree <sup>4</sup>. Finally, the compiler is able to generate an executable file that can be run on our personal computers <sup>5</sup>.

## 5 Conclusions

The successful development of this compiler, as evidenced by its ability to process source code through all compilation stages and generate a functional executable, underscores the practical value of the theoretical concepts explored. The results, particularly the compiler's correct handling of various test cases encompassing arithmetic operations, conditional statements, loops, and function calls, demonstrate a tangible outcome of applying established compiler design principles.

This project effectively bridged the gap between theory and practice. The foundational theories of formal languages were crucial for implementing the lexical analyzer with regular expressions. Similarly, context-free grammars and parsing techniques like recursive descent were instrumental in constructing the syntax analyzer and generating Abstract Syntax Trees. The systematic application of semantic analysis rules, facilitated by symbol tables and the Visitor pattern, ensured the logical correctness of the code. Furthermore, the translation to Three-Address Code and subsequent generation of x86 assembly code relied heavily on understanding intermediate representations and target machine architecture.

The constraints imposed on the language design, while simplifying implementation, did not detract from the core learning objective: to understand and apply the multi-phase compilation pipeline. Each phase, from tokenization to code generation, presented unique challenges that were addressed by drawing upon specific theoretical knowledge. The development process highlighted how these

distinct theoretical components integrate to form a cohesive system capable of translating high-level human-readable code into machine-executable instructions. This experience provides a solid understanding of the complexities inherent in compiler construction and the critical role of a strong theoretical framework in navigating these complexities to achieve a working solution.

## 6 References

### References

- [1] S. GeeksforGeeks, "Types of Parsers in Compiler Design," *GeeksforGeeks*. [Online]. Available: <https://www.geeksforgeeks.org/types-of-parsers-in-compiler-design/>. [Accessed: May 27, 2025].
- [2] S. GeeksforGeeks, "Problem on LR(0) Parser," *GeeksforGeeks*. [Online]. Available: <https://www.geeksforgeeks.org/problem-on-lr0-parser/>. [Accessed: May 27, 2025].
- [3] S. GeeksforGeeks, "Semantic Analysis in Compiler Design," *GeeksforGeeks*. [Online]. Available: <https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/>. [Accessed: May 28, 2025].
- [4] S. Pgrandinetti, "How to Design Semantic Analysis," *Pgrandinetti GitHub Pages*. [Online]. Available: <https://pgrandinetti.github.io/compilers/page/how-to-design-semantic-analysis/>. [Accessed: May 28, 2025].
- [5] S. Tutorialspoint, "Compiler Design Semantic Analysis," *Tutorialspoint*. [Online]. Available: [https://www.tutorialspoint.com/compiler\\_design/compiler\\_design\\_semantic\\_analysis.htm](https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm). [Accessed: May 28, 2025].
- [6] S. GeeksforGeeks, "Intermediate Code Generation in Compiler Design," *GeeksforGeeks*. [Online]. Available: <https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/>. [Accessed: Jun. 1, 2025].
- [7] S. GeeksforGeeks, "Issues in the design of a code generator," *GeeksforGeeks*. [Online]. Available: <https://www.geeksforgeeks.org/issues-in-the-design-of-a-code-generator/>. [Accessed: Jun. 1, 2025].