**Universidad Nacional Autónoma de México**

Computer Engineering

Compilers

# COMPILERS, INTERPRETERS AND ASSEMBLERS

Students:
320206102
316255819
423031180
320117174
320340312

Group:
05

Semester:
2026 - I

Mexico City, August 2025

# Index

# 1. Introduction

The study of tools for translating programming languages, such as compilers, interpreters, and assemblers, represents an essential aspect of computer engineering. Generally, like students, it is difficult to connect the theory learned in class with its practical applications.

The purpose of this work is based on the central role these tools play in software development. They make programs run efficiently, remain portable across platforms, and can be optimized for performance. Therefore, the understanding of their internal mechanism strengthens both academic knowledge and professional skills.

For that reason, this work pretends to study concrete examples of compilers, interpreters, and assemblers to show their structure and operation, which will help us to have a clear perspective on the processes that enable programming languages to be translated and executed.

# 2. Theorical Framework

**Compiler:** It is a type of software that translates a program from a source language into an equivalent version written in a target language. During this process, it also detects and reports any errors in the original code. When the translation produces a machine-executable program, the user can run it to process inputs and produce outputs.

**Interpreter:** It is a different kind of language processor that, instead of creating a separate target program, directly executes the instructions of the source code using user input. Although programs translated into machine language usually run faster than those interpreted, interpreters often deliver more understandable error messages because they operate by executing the program step by step.

**Assembler:** It is a program that translates an assembly language program, usually generated by a compiler, into relocatable machine code. This machine code can then be linked to other object files and libraries to create the final executable program. [1]

# 3. Development

## 3.1. Compilers

### 3.1.1. Rust

**What input does it require?**
The Rust compiler, `rustc`, begins its work with the Rust source code itself. This code is usually contained in one or more `.rs` files and written according to the Rust

language grammar. The `rustc` executable call may be indirect through the use of `cargo build`.

In addition to the program files, `rustc` also requires a variety of configuration options provided at the command line. These options determine how the compilation will proceed, such as whether to optimize for speed or size, whether to include debugging information, or which platform or architecture should be targeted [10].

**The main elements of its grammar**
**Basic Blocks:** These are the fundamental units in MIR's (**Mid-level Intermediate Representation**) control-flow graph (CFG). Each **basic block** comprises:

- **Statements**: sequential actions with exactly one successor (i.e., they always proceed to the next statement).
- **Terminators**: the concluding action in a block, capable of branching to multiple successors (e.g., `goto`, `if`, `return`).

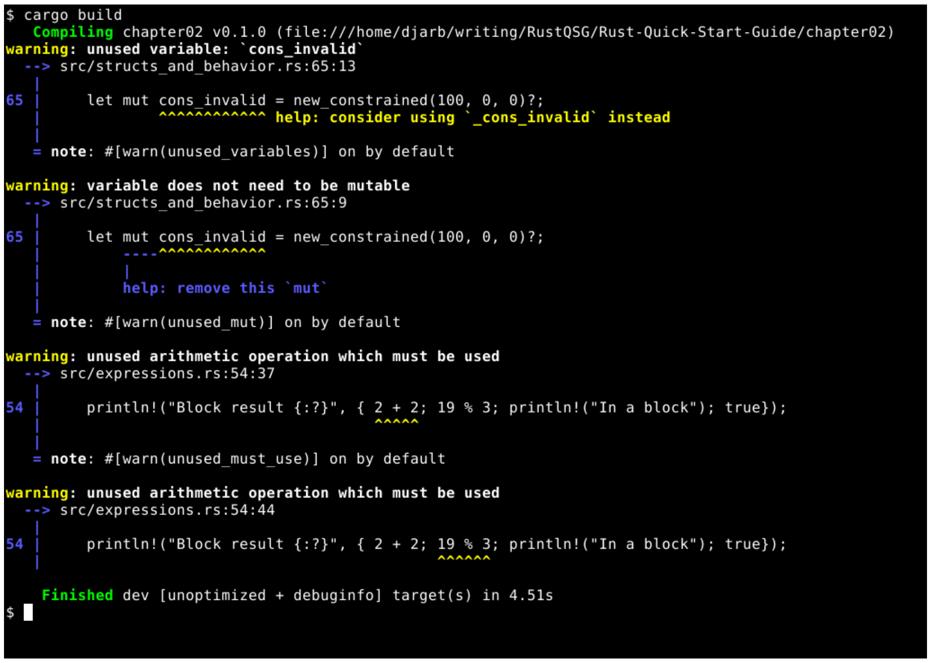
**Locals:** These refer to memory locations on the stack (conceptually). They include function arguments, local variables, and temporary values. Identified by index-like names with a leading underscore—e.g., `_1`. The special `_0` is reserved for the function's return place [9].

**The intermediate code it generates**
Like most compilers, `rustc` does not work directly on raw source code for analysis and optimization. Instead, it constructs a series of **intermediate representations (IRs)**, each tailored for different compilation tasks. These IRs gradually transform the human-readable program into forms that are easier for the compiler to check, optimize, and eventually translate into machine code:

- **Token Stream**: The lexer converts raw text into a sequence of tokens. This token stream is more structured and easier for the parser to handle than plain source code.
- **Abstract Syntax Tree (AST)**: Built from the token stream, the AST represents the program almost exactly as written by the user. It enables initial syntactic checks, such as verifying that types or expressions appear in valid contexts.
- **High-level IR (HIR)**: A desugared form of the AST, where implicit constructs (like some omitted lifetimes) are made explicit. HIR is still close to the source but is structured in a way that makes **type checking** more manageable.
- **Typed HIR (THIR)**: A typed and further desugared version of HIR. Here, operations such as method calls and implicit dereferences are made explicit. THIR serves as the main input for lowering into MIR.

- **Middle-level IR (MIR)**: A central IR in `rustc`, MIR represents the program as a **control-flow graph (CFG)**. Each basic block contains simple typed statements (like assignments and computations) with terminators that define control flow (branches, calls, drops). MIR is used for **borrow checking**, **dataflow analyses** (e.g., detecting uninitialized values), **constant evaluation** (via Miri), and early optimizations. Because it is still generic (pre-monomorphization), MIR allows analyses and transformations to be more efficient than on fully expanded code.

- **LLVM IR**: Once analyses and optimizations are done, MIR is lowered to **LLVM IR**, a typed assembly-like intermediate language widely used by compilers. LLVM IR serves as the input to LLVM's powerful optimization and code generation pipeline, ultimately producing platform-specific machine code (such as ELF executables, Windows binaries, or WebAssembly). Alternative backends like Cranelift or GCC can also be used in place of LLVM [10].



```
$ cargo build
   Compiling chapter02 v0.1.0 (file:///home/djarb/writing/RustQSG/Rust-Quick-Start-Guide/chapter02)
warning: unused variable: `cons_invalid`
  --> src/structs_and_behavior.rs:65:13
   |
65 |     let mut cons_invalid = new_constrained(100, 0, 0)?;
   |             ^^^^^^^^^^^^ help: consider using `_cons_invalid` instead
   |
   = note: #[warn(unused_variables)] on by default

warning: variable does not need to be mutable
  --> src/structs_and_behavior.rs:65:9
   |
65 |     let mut cons_invalid = new_constrained(100, 0, 0)?;
   |         ----^^^^^^^^^^^^
   |         |
   |         help: remove this `mut`
   |
   = note: #[warn(unused_mut)] on by default

warning: unused arithmetic operation which must be used
  --> src/expressions.rs:54:37
   |
54 |     println!("Block result {:?}", { 2 + 2; 19 % 3; println!("In a block"); true});
   |                                     ^^^^^
   |
   = note: #[warn(unused_must_use)] on by default

warning: unused arithmetic operation which must be used
  --> src/expressions.rs:54:44
   |
54 |     println!("Block result {:?}", { 2 + 2; 19 % 3; println!("In a block"); true});
   |                                            ^^^^^^

    Finished dev [unoptimized + debuginfo] target(s) in 4.51s
$ █
```

This screenshot of a terminal window shows the result of running the command `cargo build` in a Rust project. The output includes several compiler warnings, such as unused variables, unnecessary use of reserved keywords, and unused arithmetic operations. Each warning is highlighted with file paths, line numbers, and suggested fixes. At the end of the process, the compiler indicates that the build finished successfully in development mode with the message *"Finished dev [unoptimized + debuginfo] target(s) in 4.51s"* [7].

### 3.1.2. Javac

**What input does it require?**

javac processes Java source files with a .java extension, which must follow the Java

Language Specification. These files can represent individual classes, nested classes, or interfaces. Java's modular system allows organizing code into packages or modules, and the compiler relies on the classpath or module-path to locate all referenced types and dependencies. [14][13].

**The main elements of its grammar**
Lexical structure: establishes the fundamental components of the language, such as:

- **Identifiers:** `unlimited-length sequence of Java letters and Java digits, the first of which must be a Java letter.`
- **Reserved keywords:** `ReservedKeyword like do, default, continue, if, float, etc. or ContextualKeyword like uses, opens, permits, when, etc.`
- **Literals:** `integers, floating-point numbers, booleans, characters, strings, and the null value.`
- **Operators:** `=, >, >, +, -, *, /, !, !=, etc.`
- **Separators:** `(,), ,, [,], ;, ., @, ..., :: and ,.`

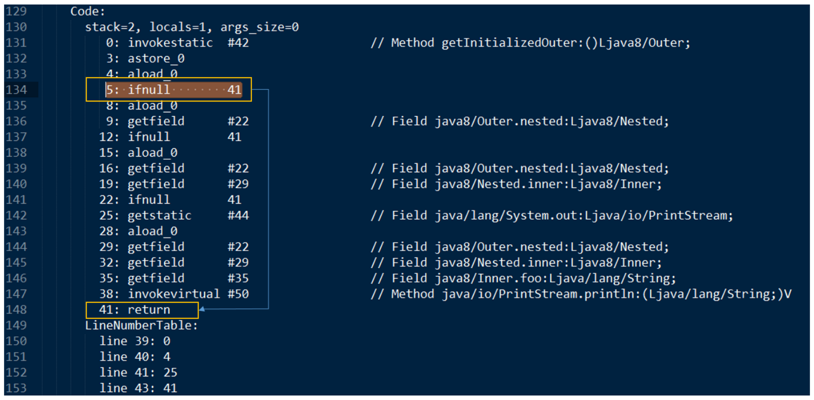
Grammatical structures: The grammar handles declarations, control statements, method definitions, class/interface declarations, and incorporates generics, annotations, exception handling, lambdas, and module statements. [13].

**The intermediate code it generates**
The Java source code (.java files) is compiled into bytecode (.class files) using the Java compiler (javac). This bytecode comprises compact, platform-independent instructions that the Java Virtual Machine (JVM) can interpret or compile just-in-time into native code. These .class files represent the intermediate code in the Java execution flow. Each class, including inner classes, results in its own .class file. This bytecode enables Java's cross-platform capabilities and is a key piece in the Write Once, Run Anywhere philosophy. [15][20].

**Results**
The next screenshot illustrates a real-world example of Java bytecode obtained using javap -c.

The sequence includes: Stack operations like iconst2, istore1, iload1, which manage constant loading and variable storage, Control flow bytecode, such as ificmpge, iinc, and goto, used to implement loops and conditional branching, and Method invocations, such as getstatic and invokevirtual, for performing console output or other method calls.

This snippet is a clear depiction of the intermediate code produced by the Java compiler (javac). Instead of machine code, Java generates bytecode, which can be executed or optimized by any Java Virtual Machine (JVM). This layer ensures portability and forms the basis for runtime efficiency and platform independence [8].



```
Code:
  stack=2, locals=1, args_size=0
     0: invokestatic  #42                 // Method getInitializedOuter:()Ljava8/Outer;
     3: astore_0
     4: aload_0
     5: ifnull        41
     8: aload_0
     9: getfield      #22                 // Field java8/Outer.nested:Ljava8/Nested;
    12: ifnull        41
    15: aload_0
    16: getfield      #22                 // Field java8/Outer.nested:Ljava8/Nested;
    19: getfield      #29                 // Field java8/Nested.inner:Ljava8/Inner;
    22: ifnull        41
    25: getstatic     #44                 // Field java/lang/System.out:Ljava/io/PrintStream;
    28: aload_0
    29: getfield      #22                 // Field java8/Outer.nested:Ljava8/Nested;
    32: getfield      #29                 // Field java8/Nested.inner:Ljava8/Inner;
    35: getfield      #35                 // Field java8/Inner.foo:Ljava/lang/String;
    38: invokevirtual #50                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    41: return
  LineNumberTable:
    line 39: 0
    line 40: 4
    line 41: 25
    line 43: 41
```

## 3.2. Interpreters

### 3.2.1. PHP

#### What input does it require?

The PHP interpreter requires source code files using the .php file extension as input. This code may be written standalone, or it may be embedded in HTML docs. However, in the case of mixed input, it is only the portions between the tags <?php or <? that are interpreted as PHP. Everything outside the programming delimiters is output without being interpreted. [16]

Execution occurs in various environments. For example, it can be executed on the web using a web server such as Apache (or more recently Nginx, often with PHP-FPM), executed via a command line interface (CLI) for standalone scripts, or executed in an interactive shell (php -a) offering a convenience for testing small pieces of code while declaratively commenting commands.

#### The main elements of its grammar

PHP defines its syntax and semantics with the following constructs: Basic Syntax and Delimiters - PHP code is written in between and tags, while shorthand echo tags can be used for inline output. Anything written outside of these delimiters is unmodified and sent straight to the output stream.

Data Types - PHP is a very rich set of types: Scalar types: bool, int, float, string. Special types: null, resource. Compound types: array, object, callable, iterable. Since PHP 8.0, union types can be used to allow one variable or return value can have a number of possible types (e.g. int|string)

Variables and Comments - Variables start with the $ symbol, and do not require any explicit type, and are case-sensitive in regard to names. There are three forms of comments: C-style /* ... */, single-line //, and shell-style

Control Structures - Control structures for PHP are conditional statements (if, elseif, and else), the ternary operator (?:), and null coalescing operator (??). A switch statement can be used for multiple select, or optionally use the match .expression.added in PHP 8.0.

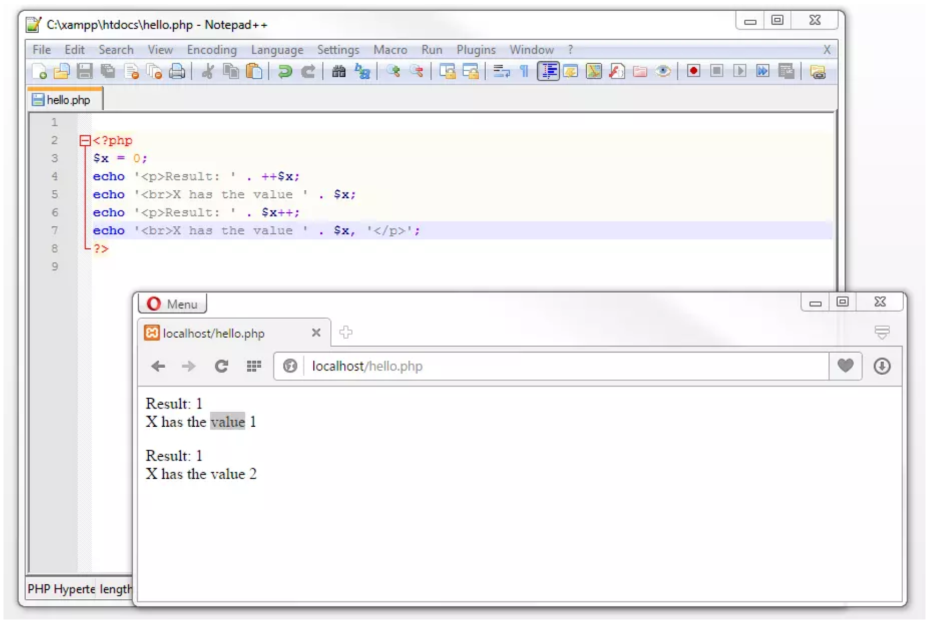
Functions - These can be user-defined or built-in. Anonymous functions, or closures, were introduced in PHP 5.3, followed by arrow functions in PHP 7.0. PHP 8.0 for functions expanded capabilities further with named parameters, typed parameters, strict typing, and return types

Generators - Generators were introduced in PHP with the yield keyword.

## The intermediate code it generates

The PHP source code you have written will never execute as just plain text, it is always going to be executed by the Zend Engine through a compilation and execution pipeline. The source code goes through lexical analysis and parsing, which means it is tokenized, and the tokens are then represented as a parse tree that represents the syntactic structure of your source code. The parse tree is compiled down to Zend opcodes, the intermediate (lower level) instruction set for PHP's virtual machine (VM). The engine then executes the Zend opcodes in sequence to execute the behavior of the program. Newer implementations of PHP also include an Opcache extension, which caches compiled Zend opcodes in memory, making copies of the opcodes, and improves efficiency by not recompiling on subsequent executions. So, the intermediate representation of PHP programs is the Zend opcodes, basically the same as bytecode in other interpreted languages, and actually the instructions executed by the runtime.

## Results


C:\xampp\htdocs\hello.php - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
hello.php
PHP Hyperte length
Menu
localhost/hello.php
localhost/hello.php

```php
<?php
$x = 0;
echo '<p>Result: ' . ++$x;
echo '<br>X has the value ' . $x;
echo '<p>Result: ' . $x++;
echo '<br>X has the value ' . $x, '</p>';
?>
```

```
Result: 1
X has the value 1

Result: 1
X has the value 2
```

This screenshot captures the running of a PHP script in a local server environment (localhost). The script shows the distinction between pre-increment (++$x) and post-increment ($x++). The first instance shows that $x is pre-incremented (it will be evaluated first), therefore the return from the expression is 1, and $x's value is also 1. The second shows that the value of $x is being used first (resulting in 1), and the variable is then incremented, leaving it with a value of 2. The output shows the different ways PHP handles the pre and post increment expressions with their particular order of evaluation in mind.

### 3.2.2. JavaScript

**What input does it require?**
JavaScript source code is a sequence of Unicode characters that the engine reads from left to right and breaks down into tokens such as identifiers, keywords, literals, and operators.The input's a piece of ECMAScript code, which is processed and checked according to lexical and syntactic grammar rules [11]

**The main elements of its grammar**

- **Lexical Grammar** : Breaks code into chunks into tokens: identifiers, keywords, literals, punctuators, white space, line terminators, and comments.

- **Data Types**: Built-in types include:

- **Primitives:** Undefined, Null, Boolean, Number, BigInt, String, Symbol.

- **Objects:** all other structures, including arrays, functions, dates, and user-defined objects.

- **Variables and Comments:** You declare variables with var, let, or const. They are super flexible with type.

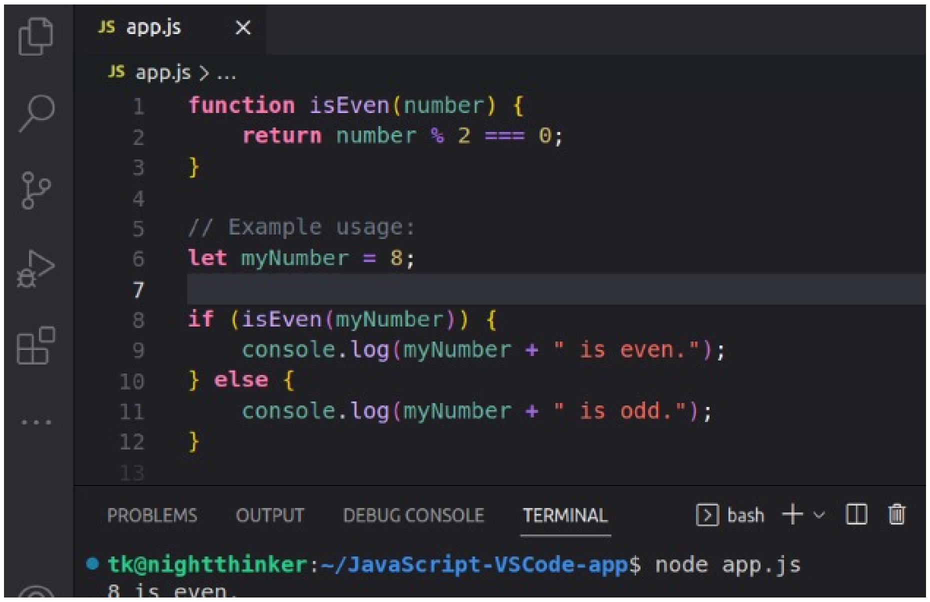
  Comments can be just a quick note(//) or a longer explanation (/* ... */)

- **Control Structures:** JavaScript provides if, else, switch, loops (for, while, do...while), iteration (for...in, for...of), and exception handling (try...catch, throw).

- **Functions and Classes:** Functions can be declared in a few ways, like using the regular function keyword, as arrow functions () =>, async functions, or generators. Classes, introduced in ECMAScript 2015, are declared with the class keyword.

  **The intermediate code it generates** JavaScript is parsed into an Abstract Syntax Tree before execution. Modern engines then compile the AST into internal intermediate representations before execution. Hot code paths are even better by turning them into machine code by Just-In-Time compilers. The intermediate code's exact details are up to how it's implemented and the ECMAScript spec doesn't lay it all out.

**Results**



```
function isEven(number) {
    return number % 2 === 0;
}

// Example usage:
let myNumber = 8;

if (isEven(myNumber)) {
    console.log(myNumber + " is even.");
} else {
    console.log(myNumber + " is odd.");
}
```

```
tk@nightthinker:~/JavaScript-VSCode-app$ node app.js
8 is even.
```

JavaScript code executed in Visual Studio Code using Node.js, showing the evaluation of an even number.

## 3.3. Assemblers

### 3.3.1. PyTeal

**What input does it require?**
PyTeal requires Python code written using the PyTeal library, which acts as an expressive, high-level binding to Algorand's low-level TEAL assembly. This code must run under **Python 3.10 or later** [2].

Within this Python environment, developers use PyTeal's expressions (`Expr` types), control-flow constructs, transaction and global fields, and optional ABI types to construct smart contract logic.

In essence, PyTeal's input is a Python script utilizing PyTeal-specific constructs, structured to build smart contract logic via the provided API.

**The main elements of its grammar**
**Expressions (`Expr`):** The foundation of PyTeal logic; they include binary/unary

operations, comparisons like `Eq`, `And`, `Or`, arithmetic (`Add`, `Div`), byte/string operations (`Bytes`, `Concat`), and cryptographic utilities (`Ed25519Verify`) [3].

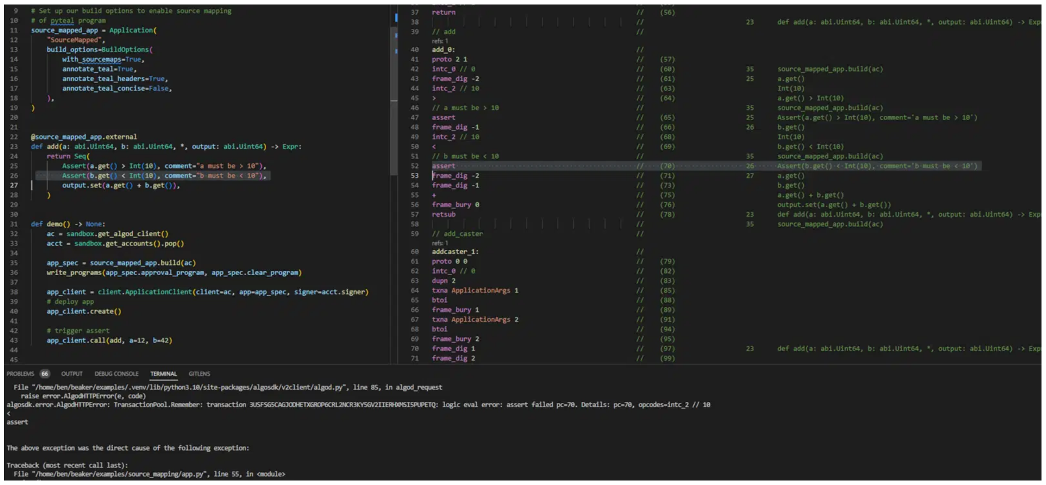
### TEAL data access

- **Transaction & global fields** (`Txn.*`, `Global.*`) such as `Txn.amount()`, `Txn.receiver()`, `Global.group_size()`, and others provide contextual data [3].
- **State access APIs** (`App.globalGet`, `App.localPut`, `App.globalGetEx`) allow reading/writing to on-chain state and optionally returning `MaybeValue` objects for safer access [6].

**The intermediate code it generates**
When you invoke `compileTeal(...)` in PyTeal, it compiles the Python-based expressions into **TEAL code**—Algorand's low-level, stack-based assembly language. This includes directives such as `#pragma version`, opcodes like `txn`, `int`, `==`, logical opcodes, and control flow markers.

This transformation works by translating all high-level constructs—expressions, subroutines, state access, ABI routines—into sequences of TEAL opcodes and directives that the Algorand Virtual Machine can execute [4]. If source mapping is enabled, PyTeal also injects comments into the TEAL output to link back to the original Python expressions [5].

Thus, TEAL acts as the intermediate representation: it is the precise, executable assembly code produced by PyTeal before it is sent to the blockchain.



```python
# Set up our build options to enable source mapping
# of pyteal program
source_mapped_app = Application(
    "SourceMapped",
    build_options=BuildOptions(
        with_sourcemaps=True,
        annotate_teal=True,
        annotate_teal_headers=True,
        annotate_teal_concise=False,
    ),
)


@source_mapped_app.external
def add(a: abi.Uint64, b: abi.Uint64, *, output: abi.Uint64) -> Expr:
    return Seq(
        Assert(a.get() > Int(10), comment="a must be > 10"),
        Assert(b.get() < Int(10), comment="b must be < 10"),
        output.set(a.get() + b.get()),
    )


def demo() -> None:
    ac = sandbox.get_algod_client()
    acct = sandbox.get_accounts().pop()

    app_spec = source_mapped_app.build(ac)
    write_programs(app_spec.approval_program, app_spec.clear_program)

    app_client = client.ApplicationClient(client=ac, app=app_spec, signer=acct.signer)
    # deploy app
    app_client.create()

    # trigger assert
    app_client.call(add, a=12, b=42)
```

```
return                      //    (56)
                            //             23    def add(a: abi.Uint64, b: abi.Uint64, *, output: abi.Uint64) -> Exp
// add                      //
refs: 1
add_0:                      //
proto 2 1                   //    (57)
intc_0 // 0                 //    (60)     35    source_mapped_app.build(ac)
frame_dig -2                //    (61)     25    a.get()
intc_2 // 10                //    (63)           Int(10)
>                           //    (64)           a.get() > Int(10)
// a must be > 10           //             35    source_mapped_app.build(ac)
assert                      //    (65)     25    Assert(a.get() > Int(10), comment='a must be > 10')
frame_dig -1                //    (66)     26    b.get()
intc_2 // 10                //    (68)           Int(10)
<                           //    (69)           b.get() < Int(10)
// b must be < 10           //             35    source_mapped_app.build(ac)
assert                      //    (70)     26    Assert(b.get() < Int(10), comment='b must be < 10')
frame_dig -2                //    (71)     27    a.get()
frame_dig -1                //    (73)           b.get()
+                           //    (75)           a.get() + b.get()
frame_bury 0                //    (76)           output.set(a.get() + b.get())
retsub                      //    (78)     23    def add(a: abi.Uint64, b: abi.Uint64, *, output: abi.Uint64) -> Exp
                            //             35    source_mapped_app.build(ac)
// add_caster               //
refs: 1
addcaster_1:                //
proto 0 0                   //    (79)
intc_0 // 0                 //    (82)
dupn 2                      //    (83)
txna ApplicationArgs 1      //    (85)
btoi                        //    (88)
frame_bury 1                //    (89)
txna ApplicationArgs 2      //    (91)
btoi                        //    (94)
frame_bury 2                //    (95)
frame_dig 1                 //    (97)     23    def add(a: abi.Uint64, b: abi.Uint64, *, output: abi.Uint64) -> Exp
frame_dig 2                 //    (99)
```

```
  File "/home/ben/beaker/examples/.venv/lib/python3.10/site-packages/algosdk/v2client/algod.py", line 85, in algod_request
    raise error.AlgodHTTPError(e, code)
algosdk.error.AlgodHTTPError: TransactionPool.Remember: transaction 3USFSG5CAGJODHETXGROP6CRL2NCR3KY5GV2IIERHXMSI5PUPETQ: logic eval error: assert failed pc=70. Details: pc=70, opcodes=intc_2 // 10
<
assert


The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "/home/ben/beaker/examples/source_mapping/app.py", line 55, in <module>
```

The figure shows the development and debugging of a PyTeal smart contract with source mapping enabled. The PyTeal code defines a simple `add` function that asserts two conditions ($a \geq 10$ and $b < 10$) before performing an addition. On the right, the compiled TEAL intermediate representation is displayed, aligned with the original code for traceability [12].

### 3.3.2. NASM

**What input does it require?**
NASM accepts assembly source code files like .asm extension. The code is structured with instructions, directives, labels, data definitios, etc. The input must conform to NASM's syntax rules and is case-insensitive for opcodes, though identifiers are case-sensitive [17].

**The main elements of its grammar**
NASM grammar is not a full context-free grammar like higher-level languages, but it has syntactic and lexical rules defined for assembling valid instructions.

- **Directives:** `commands to the assembler itself, rather than the processor, like DW (define word), EQU (assign a value to a symbolic name) and SECTION (defines code and data sections).`
- **Expressions:** `combine constants, variables, and operators to produce a value. NASM supports logical operators (AND, OR, XOR), bitwise operators and arithmetic operators (+, -, *, /).`
- **Constants:** `such as strings, character, floating-point or numeric constants`
- **Instructions:** `tell the processor what to do, examples include JMP (jump to a new location), CMP (compare), ADD (addition), SUB (subtraction) and MOV (move data).`
- **Labels:** `symbolic names for memory addresses or code locations` [18].

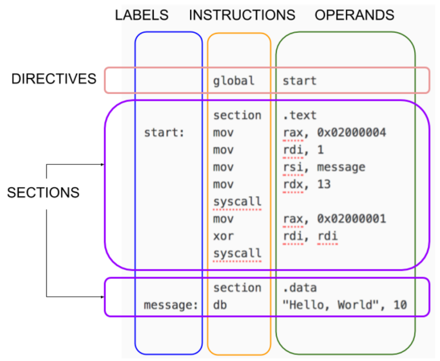
**The intermediate code it generates**
NASM generates intermediate or object code files based on the specified options. By default, it derives the output file name from the input source file, replacing the .asm extension with one appropriate to the target format (e.g., .obj for Microsoft, .o for Unix systems, .bin for raw binaries). If the file already exists, NASM overwrites it, unless it conflicts with the input name, in which case nasm.out is used.

Users can explicitly control this behavior using the -o option to define the output file name, and the -f option to specify the file format (e.g., bin, elf, obj).

These options ensure flexibility in producing intermediate files suitable for linking or direct execution.

Example:
nasm -f bin program.asm -o program.com
nasm -f bin driver.asm -o driver.sys [19].

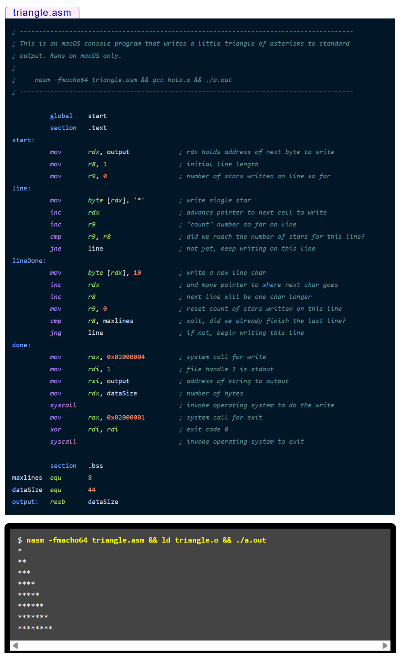
## Results



```
         global    start
         section   .text
start:   mov       rax, 0x02000004
         mov       rdi, 1
         mov       rsi, message
         mov       rdx, 13
         syscall
         mov       rax, 0x02000001
         xor       rdi, rdi
         syscall
         section   .data
message: db        "Hello, World", 10
```

The image illustrates the layout of an ELF object file created by NASM. We can see separate segments for code (.text), data (.data), read-only data (.rodata), along with symbol tables and relocation entries. This structure is typical of files with a .o extension when you assemble with nasm -f elf... [21].

One example of that is:



triangle.asm

```
; ----------------------------------------------------------------------------------------
; This is an macOS console program that writes a little triangle of asterisks to standard
; output. Runs on macOS only.
;
;     nasm -fmacho64 triangle.asm && gcc hola.o && ./a.out
; ----------------------------------------------------------------------------------------

          global    start
          section   .text
start:
          mov       rdx, output             ; rdx holds address of next byte to write
          mov       r8, 1                   ; initial line length
          mov       r9, 0                   ; number of stars written on line so far
line:
          mov       byte [rdx], '*'         ; write single star
          inc       rdx                     ; advance pointer to next cell to write
          inc       r9                      ; "count" number so far on line
          cmp       r9, r8                  ; did we reach the number of stars for this line?
          jne       line                    ; not yet, keep writing on this line
lineDone:
          mov       byte [rdx], 10          ; write a new line char
          inc       rdx                     ; and move pointer to where next char goes
          inc       r8                      ; next line will be one char longer
          mov       r9, 0                   ; reset count of stars written on this line
          cmp       r8, maxlines            ; wait, did we already finish the last line?
          jng       line                    ; if not, begin writing this line
done:
          mov       rax, 0x02000004         ; system call for write
          mov       rdi, 1                  ; file handle 1 is stdout
          mov       rsi, output             ; address of string to output
          mov       rdx, dataSize           ; number of bytes
          syscall                           ; invoke operating system to do the write
          mov       rax, 0x02000001         ; system call for exit
          xor       rdi, rdi                ; exit code 0
          syscall                           ; invoke operating system to exit

          section   .bss
maxlines  equ       8
dataSize  equ       44
output:   resb      dataSize
```

```
$ nasm -fmacho64 triangle.asm && ld triangle.o && ./a.out
*
**
***
****
*****
******
*******
********
```

For the triangle area formula (area = (base * height) / 2), the compiler generates intermediate code in a machine-independent form, often as three-address instructions: t1 = base * height; t2 = t1 / 2; area = t2;. This stage acts as a bridge between source code and machine code, making optimization and translation to different architectures easier [1].

## 4. Conclusions

The objectives we set were satisfactorily achieved. By looking at real-life examples of compilers (Rustc), interpreters (PHP and JavaScript), and assemblers (PyTeal and NASM), we got a chance to dig into how they work, their structure, inputs, grammar, and the intermediate code generated. This helped us link up the theory with how it actually works in real life, making it easier to get how programming languages turn into action and run.

So, the work fulfilled its purpose of providing a clearer perspective on the role of these tools in the software development process and showed just how crucial they are for learning and in professional practice.

## Referencias

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools.* 2.ª ed. Pearson/Addison Wesley, 2006. ISBN: 9780321486813.

[2] Algorand. *Install PyTeal.* PyTeal Documentation, accessed Aug. 2025. 2025. URL: `https://pyteal.readthedocs.io/en/stable/installation.html`.

[3] Algorand. *PyTeal Package.* PyTeal Documentation, accessed Aug. 2025. 2025. URL: `https://pyteal.readthedocs.io/en/v0.9.0/api.html`.

[4] Algorand. *PyTeal: Python Language Binding for Algorand Smart Contracts.* Version 0.6.0, PDF, Aug. 21, 2020. 2020. URL: `https://pyteal.readthedocs.io/_/downloads/en/v0.6.0/pdf/`.

[5] Algorand. *Source Mapping HowTo.* PyTeal Documentation, accessed Aug. 2025. 2025. URL: `https://pyteal.readthedocs.io/en/stable/sourcemap.html`.

[6] Algorand. *State Access and Manipulation.* PyTeal Documentation, accessed Aug. 2025. 2025. URL: `https://pyteal.readthedocs.io/en/latest/state.html`.

[7] Daniel Arbuckle. *Compiling our project*. in *Rust Quick Start Guide*, Packt Publishing. 2018. URL: https://subscription.packtpub.com/book/programming/9781789616705/1/ch01lvl1sec04/compiling-our-project.

[8] SAP Community. *An example to use javap to analyze Java source code*. 2023. URL: https://community.sap.com/t5/technology-blog-posts-by-sap/an-example-to-use-javap-to-analyze-java-source-code/ba-p/13275566.

[9] The Rust Project Developers. *Key MIR vocabulary*. Rust Compiler Development Guide. Ago. de 2025. URL: https://rustc-dev-guide.rust-lang.org/mir/index.html?highlight=grammar#key-mir-vocabulary.

[10] The Rust Project Developers. *Overview of the compiler*. Rust Compiler Development Guide. Ago. de 2025. URL: https://rustc-dev-guide.rust-lang.org/overview.html.

[11] ECMA International. *ECMAScript® 2024 Language Specification (ECMA-262, 14th Edition)*. Accessed: 2025-08-21. 2024. URL: https://262.ecma-international.org/.

[12] Adam Kenyon. *PyTeal 0.24.0: Sourcemapping!* Algorand Developer Program. Mar. de 2023. URL: https://developer.algorand.org/articles/pyteal-sourcemapping/.

[13] Oracle. *Java SE specifications*. 2025. URL: https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html.

[14] Oracle. *Javac*. 2025. URL: https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html.

[15] Oracle. *The java® virtual machine specification*. 2025. URL: https://docs.oracle.com/javase/specs/jvms/se21/html/.

[16] PHP Documentation Group. *PHP: Language Reference*. Accessed: 2025-08-21. 2025. URL: https://www.php.net/manual/en/langref.php.

[17] NASM Development Team. *NASM Macro Language Reference*. Algorand Developer Program. 2024. URL: https://www.nasm.us/doc/nasmdoc1.html.

[18] NASM Development Team. *NASM Macro Language Reference*. 2024. URL: https://www.nasm.us/doc/nasmdoc3.html.

[19] NASM Development Team. *NASM Macro Language Reference*. 2024. URL: https://www.nasm.us/doc/nasmdoc2.html.

[20] TechnogeeksCS. *How is Java Bytecode Generated?* 2025. URL: https://technogeekscs.com/java-bytecode/.

[21] Loyola Marymount University. *NASM Tutorial*. 2015. URL: https://cs.lmu.edu/~ray/notes/nasmtutorial/.