Universidad Nacional Autónoma de México

Computer Engineering.

Compilers

PRACTICE NO. 1

STUDENT:
320254536
315293728
423108596
320246881
320250648

Group:
5
Semester:
2026-I

México, CDMX. August 2025

Contents

1 Introduction

Throughout this practice, we conducted research to gain a deeper understanding of the composition and functioning of different types of language processors in computer systems that were first discussed during class sessions. This work aims to highlight the importance of understanding the existing approaches with the intention of building our own implementation in the future, as well as improving the efficiency with which we use computer systems [1].

In other words, the main objective of this course on compilers is to apply development techniques and tools for interpreters, compilers, and other translators [2]. Therefore, this research will lay the foundation for a better understanding of the course, establishing a detailed understanding of different languages that we've used during our university career. Furthermore, the reason why we study compilers in computer engineering is to become better programmers as well as contributing to the creation of new tools for debugging, translating, or even creating new languages [3]. Furthermore, it can also help in recognizing the patterns or main elements each one of them need in order to work and be able to draw comparisons between them.

2 Theoretical Framework

Programming languages cannot be directly executed by hardware; instead, they require translation into machine code. There are three major categories of language translators: compilers, interpreters, and assemblers. The process of generative translation from a high-level, human-readable language into a more easily executed form is called compilation, and the programs that perform this translation are known as compilers [1]. In other words, compilation is the process of converting textual source code in a programming language into machine code, which is stored in a file known as an executable file (binary file) [4].

The complete structure of a compiler, like we've seen in class, is the following:

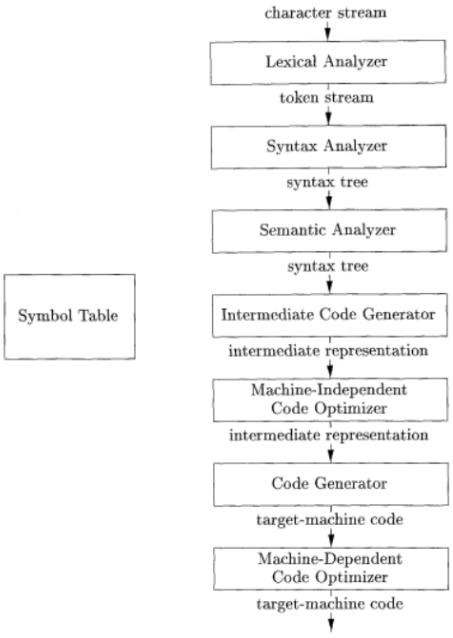


Figure 1: Phases of a compiler [5]

An interpreter, on the other hand, is "another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations

specified in the source program on inputs supplied by the user" [1]. This approach allows immediate execution of programs without generating a separate executable file, in contrast to compilers.

Assemblers are also essential components of language-processing systems. As described by Aho, Lam, Sethi, and Ullman, "the compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug. The assembly language is then processed by a program called an assembler that produces relocatable machine code as its output" [1]. In this way, assemblers serve as translators that bridge human-readable assembly code and executable machine code.

Another difference between a compiler and an interpreter is that an interpreter can perform static analysis (analysis performed during the execution of a program) and static analysis (analysis before the execution of a program), whereas a compiler only performs static analysis [6].

Another important concept, which has even been addressed in previous subjects, is grammar, since it describes the hierarchical structure of programs; in this case, we are especially interested in context-free grammar because many programming languages rely on it. This type of grammar is composed of terminal symbols or tokens , where a token is a pair formed by a token name, an abstract symbol that represents a kind of lexical unit, such as a keyword or an identifier, and an optional attribute value [6]. Each token is associated with a pattern, which specifies the form its lexemes may take; for keywords the pattern corresponds directly to the sequence of characters that define them, while for identifiers or other categories the pattern can be more general and match multiple strings. So a lexeme is the actual sequence of characters in the source program that matches a pattern and is recognized as an instance of a token. Together with nonterminals, productions, and the start symbol, these elements allow compilers to define formal rules that reduce ambiguities and implement algorithms for determining how a string of terminals can be generated by a grammar, read and classify the input characters of the source program, report syntax errors and recovery strategies [6].

3 Development

3.1 Compiler

3.1.1 C

As mentioned above, one category of language translators is compilers, which allow a program in a source language to be translated into a program in a target language [3]. For example, C is well known for translating a high-level language into native assembly language so that a machine can execute it. First of all, a compiler is just a component in a tool chain of programs used to create executables from source code. The following figure explains the common Unix system for compiling C source code [3].

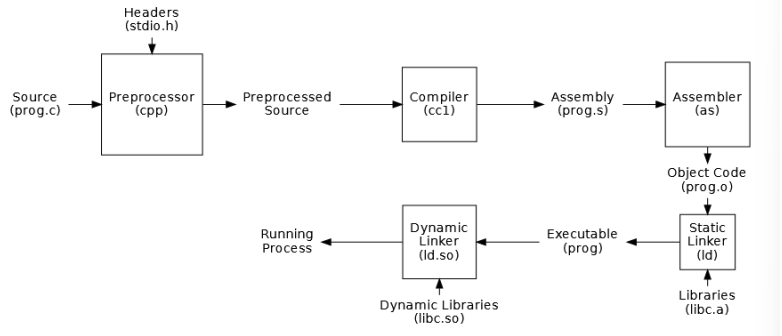


Figure 2: Compiler Toolchain [3]

In C, the input we require for the compiler is not only C source code, but also preprocessed

code, which means consuming all directives (those starting with #), expanding macros, including header files, and selecting conditionally compiled code [7]. This process is done in the first step of the compiler toolchain; in the case of GCC it transforms the text with text substitutions and file inclusions. Once the preprocessing is complete, the scanner consumes the plain text, so the program has been reduced to a sequence of tokens [8].

For the grammar of the C language, we will cover ANSI C, which is a standardized version of the C programming language that was introduced in 1989 by the American National Standards Institute (ANSI). The grammar of C is defined in terms of a context-free grammar expressed in BNF (Backus-Naur Form) with several production rules. This grammar has undefined terminal symbols integer-constant, character-constant, floating-constant, identifier, string, enumeration-constant [8], with the addition that they can be transformed into acceptable input for an automatic parse generator such as Bison or YACC. The compiler first performs a lexical analysis to recognize tokens and then uses this grammar to verify that the source code is syntactically correct; a token, for example, according to the standard, could be any of the following [9]:

1. **Keyword**: *one of*

   auto break case char const continue default
   do double else enum extern float for goto
   if inline int long register restrict return
   short signed sizeof static struct switch typedef
   union unsigned void volatile while
   _Alignas _Alignof _Atomic _Bool _Complex
   _Generic _Imaginary _Noreturn _Static_assert _Thread_local

2. **Identifier**:

   identifier-nondigit
   identifier identifier-nondigit
   identifier digit

3. **Constant**:

   integer-constant floating-constant
   enumeration-constant character-constant

4. **String literal**:

   $\text{encoding-prefix}_{\text{opt}}$ “s-char-sequence”$_{\text{opt}}$
   encoding-prefix ::= u8 u U L

5. **Punctuator**: *one of*

   [ ] ( ) { } . ->
   ++ – & * + - ~ ! / % « » < > <= >= == !=
   ^ & && || ? : ; ... = *= /= %= += -= «= »=
   &= ^= |= , # ## <% %> <: :> <:> %: %:%:

Then, the parser consumes tokens into statements and expressions, which is led by a grammar to create an abstract syntax tree (AST). This means the parser looks for patterns that match the established grammar for the C language [3], also the compiler applies semantic routines to the AST and meaning beyond syntax, then transforms to an Intermediate Representation (IR), which is an abstract assembly language in order to optimize. Finally the code generator translates the optimized IR into assembly code [3]. For the intermediate code in GCC compiler, Tree (AST) represents the program's structure; then it is lowered into GIMPLE, a three-address code that simplifies expressions [10], which is further converted into SSA (Static Single Assignment) form to enable powerful optimizations [11]; afterwards, GCC generates RTL (Register Transfer Language), a low-level IR resembling machine operations [12]; finally, this is translated into assembly code, which is assembled into machine code and linked with the existing libraries into the executable [4].

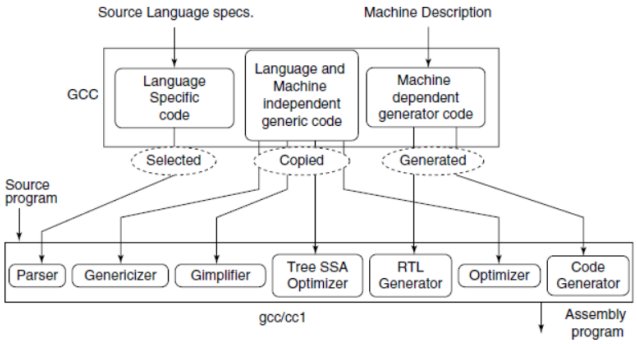


Figure 3: Real-life: Intermediate Codes of GNU gcc [13]

3.1.2 RUST

Rust is a programming language that excels at memory efficiency. It is used in several projects, such as browsers, operating systems, and web development. Rust's primary objective is to provide tools that facilitate working with the low-level details of memory management while minimizing risks.

In Rust, compilation begins with a Rust source program. You can either invoke the rustc compiler directly or let cargo, the Rust build system and package manager, handle it for you. The fundamental compilation unit in Rust is called a crate. According to the Rustc Dev Guide, "After you invoke the compiler, command-line argument parsing occurs in rustc_driver. This crate defines the compilation configuration requested by the user and passes it to the rest of the compilation process as a rustc_interface::Config"[14]

RustC uses several intermediate representations to facilitate compilation. These representations are:

- **Token Streams**: The lexer produces a stream of tokens from the source code, which are then sent to the parser.
- **Abstract Syntax Tree (AST)**: The AST is built by the parser from the stream of tokens produced by the lexer.
- **High-level Intermediate Representations (HIR)**: This is a more compiler-friendly version of the AST, but it is still close to what the user wrote.
- **Typed High-level Intermediate Representations (THIR)**: This is similar to the HIR, but it is fully typed.

- **Mid-level Intermediate Representations (MIR)**: This IR is basically a Control-Flow Graph (CFG). A CFG is a type of diagram that shows the basic blocks of a program and how control flow can go between them [14].
- **LLVM-IR**: This is the standard form of all input to the LLVM compiler.

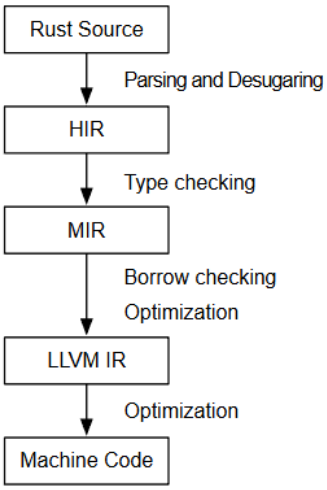


Figure 4: Rust internal compiler pipeline [15].

The following table presents the notations used by the Rust lexer and in syntax grammar snippets:

Notation	**Examples**	**Meaning**
CAPITAL	KW_IF, INTEGER_LITERAL	A token produced by the lexer
ItalicCamelCase	LetStatement, Item	A syntactical production
string	x, while, *	The exact character(s)
x?	pub?	An optional item
x*	OuterAttribute*	0 or more of x
x+	MacroMatch+	1 or more of x
xa..b	HEX_DIGIT1..6	a to b repetitions of x
Rule1 Rule2	fn Name Parameters	Sequence of rules in order
|	u8 | u16, Block | Item	Either one or another
[ ]	[b B]	Any of the characters listed
[-]	[a-z]	Any of the characters in the range
[ ]	[b B]	Any characters, except those listed
string	\n, */	Any characters, except this sequence
( )	(, Parameter)?	Groups items
U+xxxx	U+0060	A single unicode character
<text>	<any ASCII char except CR>	An English description of what should be matched
Rule suffix	IDENTIFIER_OR_KEYWORD except crate	A modification to the previous rule

Table 1: Rust syntax notation examples and meanings[16]

3.2 Interpreter

3.2.1 JavaScript

Interpreters, as previously defined, directly execute operations specified in the source program rather than producing a separate target program. In the context of JavaScript, the source code written in `.js` files is processed by a JavaScript engine (e.g., V8, SpiderMonkey, JavaScriptCore), which interprets and executes the code within the browser or runtime environment [17].

The execution flow of a Javascript program can be visualized in the diagram.

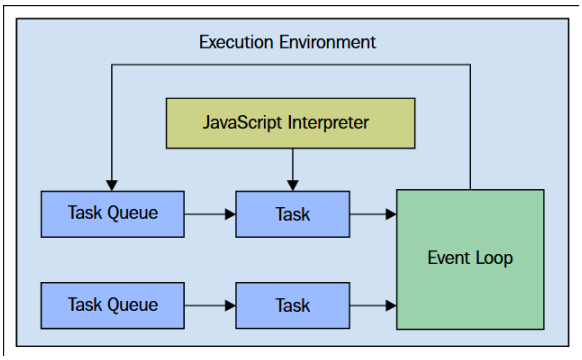


Figure 5: Javascript Execution Environment [18].

As of 2020, the majority of websites use JavaScript, and all modern web browser include JavaScript interpreters, making it the most-deployed programming language.

The input to a JavaScript interpreter is plain-text source code following the ECMAScript standard inside a file with the `.js` extension. The ECMAScript specifications further defines that the input to a program is a sequence of characters in the Unicode character set [19]. This source text can be classified into four types of code:

- **Global code**, which corresponds to the top-level body of a script;
- **Eval code**, which is the source text supplied to the built-in `eval` function;
- **Function code**, which represents the source text that defines the body and parameters of functions;
- **Module code**, which corresponds to code provided as a module body, allowing explicit imports and exports between program units.

These distinctions ensure that interpreters and engines handle code consistently depending on its context of execution. Continuing to define the different kinds of source text, the ECMAScript specifications also establish the grammar rules that determine how JavaScript programs are structured and interpreted, these are:

- **Lexical grammar**, sequence of input elements, which are tokens, line terminators, comments, or white space [19].
- **Syntactic grammar**, input elements other than white space and comments form the terminal symbols for the syntactic grammar for ECMAScript and are called ECMAScript tokens. These tokens are the reserved words, identifiers, literals, and punctuators of the ECMAScript language. Moreover, line terminators, although not considered to be tokens, also become part of the stream of input elements and guide the process of automatic semicolon insertion [19].
- **RegExp grammar**, specifies how sequences of code points are translated into regular expression patterns. Its productions start from the goal symbol Pattern, and they describe how input characters are translated into regular expressions. Productions of the lexical and RegExp grammars are distinguished by having two colons "::" as separating punctuation [19].

- **Numeric string grammar**, it has as its terminal symbols SourceCharacter, and is used for translating Strings into numeric values starting from the goal symbol StringNumericLiteral [19].

Here we can see an example of how the rule defines how additive expressions are constructed in JavaScript: either as a MultiplicativeExpressions or as another AdditiveExpression followed by a + or a - operator and a MultiplicativeExpression.

```
AdditiveExpression[Yield, Await] :
    MultiplicativeExpression[?Yield, ?Await]
    AdditiveExpression[?Yield, ?Await] + MultiplicativeExpression[?Yield, ?Await]
    AdditiveExpression[?Yield, ?Await] - MultiplicativeExpression[?Yield, ?Await]
```

Figure 6: Example of a syntactic grammar rule for JavaScript additive expressions

Modern JavaScript engines like V8 process the source code through multiple stages before execution. Initially, the code is parsed into an Abstract Syntax Tree (AST). V8's Ignition bytecode compiler then takes the AST as input and produces a stream of bytecode (BytecodeArray) along with associated metadata, that enables the interpreter to execute the code efficiently[20]. The compilation pipeline has been designed to allow background compilation: most steps are performed on a separate thread, while only the final stages AST internalization and bytecode finalization run on the main thread just before execution. Bytecode finalization involves building the final BytecodeArray object, used to execute the function, alongside associated metadata, and a SourcePositionTable which maps the JavaScript source line and column numbers to bytecode offset. This approach reduces memory overhead and keeps the main thread responsive, allowing it to handle user interactions and rendering tasks [21].

3.2.2 Python

Python is a high-level general-purpose programming language created with the intention of coding and updating programs quickly as the needs of the programmer change [22] [23]; which is why the interpreter implementation was used. It is also highly portable and can handle complex and great amounts of data easily, used widely among academia, business and science fields [22]. Python even has a list of different interpreters, such as IPython— a shell alternative to CPython, IronPython for the .NET network, Jython for Java, PyPy for enhancing performance, and others [23] .

The Python language can accept and process input from more than one source due to its flexible nature. According to the official Python documentation, they are roughly described as: a script file passed as a command-line argument, standard input, interactive typing (REPL), and strings passed to functions [24]. These are separated into the three following categories.

- **File Input:** Includes scripts (plain-text source code with one or more lines of instructions or statements inside a file with `.py` extension), modules or a string passed to the `exec()` function, which supports dynamic execution of Python code [25]. It is the most common type of input.

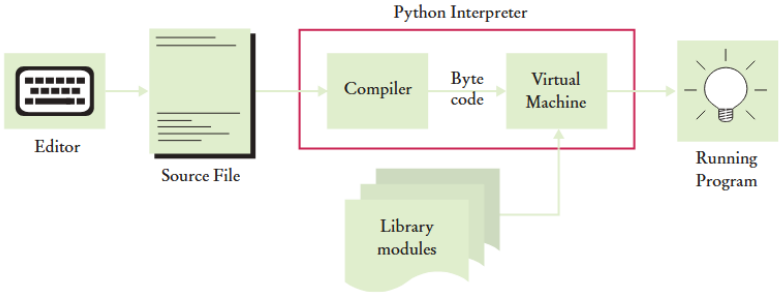


Figure 7: Workflow of a Python Program [22].

- **Interactive Input:** Happens when executing Python code in interactive mode/environment (REPL or Read-Eval-Print Loop), indicated by a chevron. It processes one line at a time with immediate results and no editing or saving [26]; more specifically, it reads in an expression, evaluates it and prints the result [27].
- **Expression Input:** Special case that treats an specified expression, not a statements via the `eval()` function, which is parsed as a expression (a condition list) [25].

Concerning Python's grammar, it's an LL(1)-based grammar [28], meaning a context-free grammar suited for top-down parsing [29] and strictly speaking, because it can be parsed by an LL(1) parser— in this case the CPython parser, even though some of their grammar rules do not follow the LL(1) grammar behavior [28]. It should be noted that the CPython parser— as of the newest version (3.9) according to the creation of this investigation, is no longer a LL(1) parser but a PEG parser in order to handle ambiguity more efficiently, the 3.8 version and earlier versions still use a LL(1) parser.

The implemented notation is a mix of EBNF (Extended Backus-Naur form) and PEG (Parsing Expression Grammar). The Python grammar notation, in particular, follows the PEP 617 [28] and Full Grammar Specification [30]. We are going to separate them into lexical grammar analysis and syntactical grammar processing.

For Python's lexical grammar analysis, the input to the parser is a stream of tokens generated by the *tokenizer* [31]. We have the subsequent.

- Statements or a logical line ends with the `NEWLINE` character (line break). If there are more than one on the same line they can be separated by a semicolon (;). Can be simple (return statement) or compound (loops, function definition).
- Line joining can be done via a backslash character (\) that is not part of a string or literal, or splitting the expressions in parentheses, square brackets or curly braces.
- Leading whitespace (spaces and tabs) at the beginning of a logical line is represents the indentation level of the line.
- Symbols or identifiers are defined with `NAME`, they are tokens (basic lexical unit) of one or more characters including letters, digits or underscore (_), but they cannot start with a digit.
- Keywords are case senstivie, a set of them is `False, True, else, break, in, is, return, and, continue, def, for, try, as, not, with, async, elif, if, or, import, except`.
- Soft keywords only get triggered in specific contexts e.g. `match, case, type and _`.
- They define numerical, string and byte literals, which are notations for constant values.
- Operator tokens: +, -, *, **, /, //, %, @, «, », &, |, ¸, ˛, :=, <, >, <=, >=, ==, !=.
- Some delimiter tokens: +=, -=, *=, @=, /=, %=, &=, |=, ˆ=, «=, »=, **=, //=.
- A comment starts with a hash character (#) that is not part of a string literal, and ends at the end of the physical line. `# comment`.

For Python's synthatic grammar analysis, lookahead refers to the first, leftmost symbol in a left-to-right scan terminal of the input string [5], meaning it checks for the next and indicates if it does or does not match with an specified pattern.

- **&** followed by a symbol, token or parenthesized group indicates a positive lookahead (match required).
- **!** indicates a negative lookahead (match not required).

- | meaning PEG's ordered choice operator (parse `a`, if it fails, backtrack and parse `b`. Succeeds if either a or b succeeds) [32].
- It is composed by a sequence of rules of the form: `rule_name: expression`
  If it has a returning type for the rule, then: `rule_name[return_type]: expression`
- The colon `:` is the key symbol that signifies the end of a header and the beginning of a block.

```
simple_stmts:
| simple_stmt !';' NEWLINE # Not needed
| ';'.simple_stmt+ [';'] NEWLINE
```

- Assignments are of the form `destination = source`.
- Blocks are defined as a group of one or more statements with the help of indentation.

```
block:
| NEWLINE INDENT statements DEDENT
| simple_stmts
```

- Function definitions.

```
function_def:
|   decorators function_def_raw
|   function_def_raw
```

  The raw definition of a function should contain the indentation, the `DEF` keyword, followed by the token identifier and the parameters.

- For function parameters, the allowed combinations/orderings are specified such as regular parameters, postional only and for the args functions.
- Lists are defined between square brackets, tuples with parentheses with at least one expression and sets between curly braces with one or more expression called `star_named_expressions`.
- Some of the grammar atoms (fundamental unit blocks) are NAME, booleans, None, strings, NUMBER, tuples, lists, dictionaries (`dict`) and an ellipsis (...).
- Strings can be a regular `STRING` literal or an f-string (for dynamic strings).

```
strings:  (fstring|string)+
```

- For the if or elif structure it follows a general structure of `if` followed by the condition and joined with another block, an elif block `elif_stmt` or an `else_block`.

```
if_stmt:
| 'if' named_expression ':'  block elif_stmt | [else_block]
| 'else' ':'  block
```

- Similar to if, `while` is also followed by a condition and an `else_block`.

```
while_stmt:
| 'while' named_expression ':'  block [else_block]
```

- A for loop requires the keywords `FOR` followed by the targets, then `IN`, the condition and finally the `[TYPE_COMMENT] block [else_block]`.

As shown in Figure 7, the Python interpreter converts source code to intermediate code called bytecode, portable code or p-code [33], which then, it is used by the stack-based Python Virtual Machine (PVM) and executed, similar to how a CPU executes machine code [33] [34]. It does so by also storing the bytecode in the file cache [35] generating `.pyc` extension files inside the `__pycache__` folder [34]. Additionally, bytecode does not necessarily have to work in the diverse PVMs, neither to be stable within versions [35].

The `dis` Python module stands for disassemble [36], and it dissasembles the bytecode into a more human-readable form. Used greatly for debugging, race conditions and optimization. [37]

In some environments, the Python interpreter is automatically launched with the "Run" button or option. In others, the interpreter has to be launched explicitly [22].

3.2.3 Notes

Additionally, a feature that both interpreted languages possess is *Dynamic Typing*, set of members can be changed at runtime, and it is able to automatically determine what kind of variable it is dealing with.

3.3 Assembler

3.3.1 GAS (GNU Assembler)

The GNU assembler is part of the GNU tools software suite, its primary purpose is to assemble the output of the GNU C compiler (GCC); it can, however, be used to assemble code written in native assembly. [38]

The assembler receives the source program as input, this source program may be one or more files written in the asm syntax specified by the command line parameters, by default GAS will use AT&T asm syntax.[39]

By default, the code must be written in AT&T standard asm syntax, which follows this general grammar:

- the parameters are ordered as such:
  instruction source , destination
- symbols are defined as one or more characters chosen from the set of all letters, digits and the special characters '_ . $'
- statements end with a newline character or a line separator character (';')
- statements may begin with a label followed by a key symbol
  label: .key statement
- Comments are written in between '/*' and '*/'
- Constants are numbers (or strings) written so that their value is known without knowing the context of the program[39]

It should be mentioned that newer versions of GAS support the usage of intel syntax, this must be configured using the directive '.intel_syntax' when running GAS.

The source file will be assembled into an object file, this file is a translation of the assembly language program into numbers, the default name of this file is a.out, this may be changed in the command line parameters when invoking GAS; the .out file contains the assembled program code as well as information to help the linker program 'ld' integrate the code into an executable file. [39]

3.3.2 NASM (Netwide Assembler)

Netwide Assembler is an x86 assembler originally developed by Simon Tatham and Julian Hall, the program is designed to provide portability and modularity to asm developers, in contrast to GAS which is designed as a backend for GCC, NASM was designed from the ground up to be an assembler for user written asm code.[40]

NASM is designed to receive a source program as input, this program must be written in NASM asm syntax, this is similar to intel asm syntax but less complex, the general grammar of the assembler is as follows:

- Each source line contains some combination of this four fields:
  label: instruction destination , source ;comment
- valid symbols in labels are the set of letters, digits and the special characters '_, $, #, , ~ ., ?', however, the only symbols that may be used as the first symbol of an identifier are letters, $ may be used at the beginning of an identifier to differentiate it from a reserved word

- The instruction field may contain any machine instruction.
- NASM understands four types of constants: numeric, character, string and float.
- Comments are prefixed by the symbol ';' [40] [41]

As NASM is built for portability, it is capable of outputting in a large number of formats (specified by the -f option), by default it will output a .o object file to be used by a linker.[40]

4 Results

4.1 Compilers

To visualize the C language compilation process, we will propose a simple program, where the input for the GCC compiler will be the calculation of the area of a circle with established values. The program is named **circulo.c**.

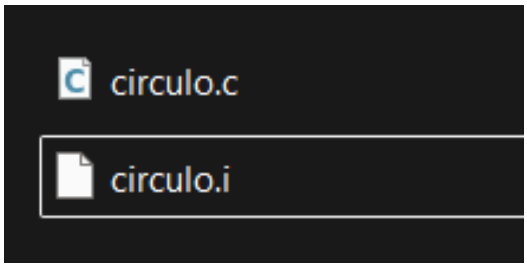
```c
#include <stdio.h>
#define PI 3.1416

int main (void)
{
float area, radio;
radio = 10;
area = PI * (radio * radio);
printf ("Area Circulo = %f\n",area);
return 0;
}
```

Figure 8: Example of a C program [42]

The first step for the lexical analyzer to receive our input is to go through processing, which would be done using the command ***$ gcc -E circulo.c -o circulo.i***. The output will be the file with the termination ".i".

```
# 3 "circulo.c"
int main (void)
{
float area, radio;
radio = 10;
area = 3.1416 * (radio * radio);
printf ("Area Circulo = %f\n",area);
return 0;
}
```

(a) Program preprocessed



(b) Output file circulo.i

Figure 9: Results of the preprocessed program `.i`

Then, with the command ***$ gcc -S circulo.c*** we create the source file with the assembly language with the previous preprocessed source code.

```
main:
        pushq   %rbp
        .seh_pushreg    %rbp
        movq    %rsp, %rbp
        .seh_setframe   %rbp, 0
        subq    $48, %rsp
        .seh_stackalloc 48
        .seh_endprologue
        call    __main
        movss   .LC0(%rip), %xmm0
        movss   %xmm0, -4(%rbp)
        movss   -4(%rbp), %xmm0
        mulss   %xmm0, %xmm0
        pxor    %xmm1, %xmm1
        cvtss2sd        %xmm0, %xmm1
        movsd   .LC1(%rip), %xmm0
        mulsd   %xmm1, %xmm0
        cvtsd2ss        %xmm0, %xmm0
        movss   %xmm0, -8(%rbp)
        pxor    %xmm0, %xmm0
        cvtss2sd        -8(%rbp), %xmm0
        movq    %xmm0, %rax
        movq    %rax, %rdx
        movq    %rdx, %xmm0
        movapd  %xmm0, %xmm1
        movq    %rax, %rdx
        leaq    .LC2(%rip), %rcx
        call    printf
        movl    $0, %eax
        addq    $48, %rsp
        popq    %rbp
        ret
        .seh_endproc
        .section .rdata,"dr"
        .align 4
```

Figure 10: Fragment of the assembly code with windows x86

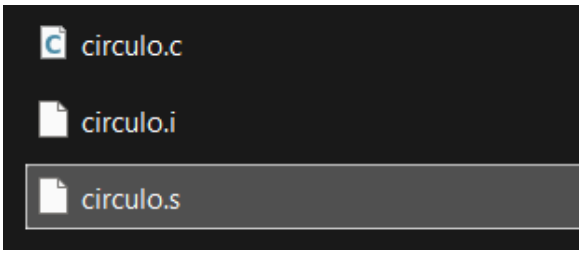


Figure 11: File circulo.s

The final stage of compilation is the linking of object files to create an executable. In practice, an executable requires many external functions from system and C run-time (crt) libraries, we can do that by writing ***$ gcc circulo.c -o circulo***

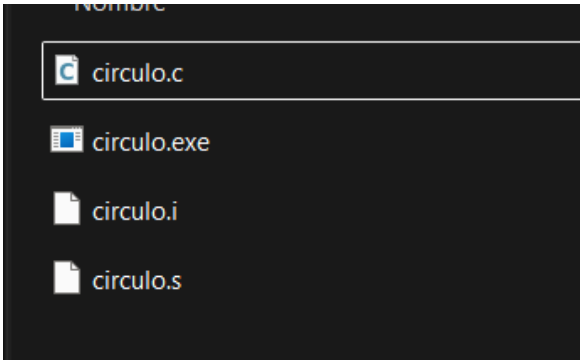


Figure 12: Final output from circulo.exe

```
PS C:\Users\jocel\OneDrive\Documentos\Semestre2026-1\Compiladores\Tarea1> .\circulo.exe
Area Circulo = 314.160004
```

Figure 13: circulo.exe

If we only write the command ***gcc program.c***, all stages of compilation: preprocessing, compilation, assembly, and linking will be performed at once, generating the executable program. [4]

In the case of Rust, we start with a '.rs' file containing a simple Hello World program:

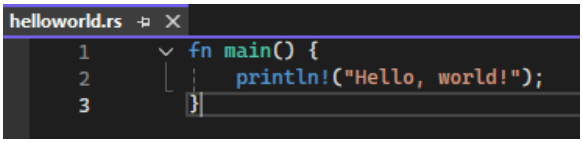


Figure 14: Helloworld.rs.

Using the -Z unpretty command, which allows us to inspect the code's internal representations, we can obtain the AST produced by the parser.

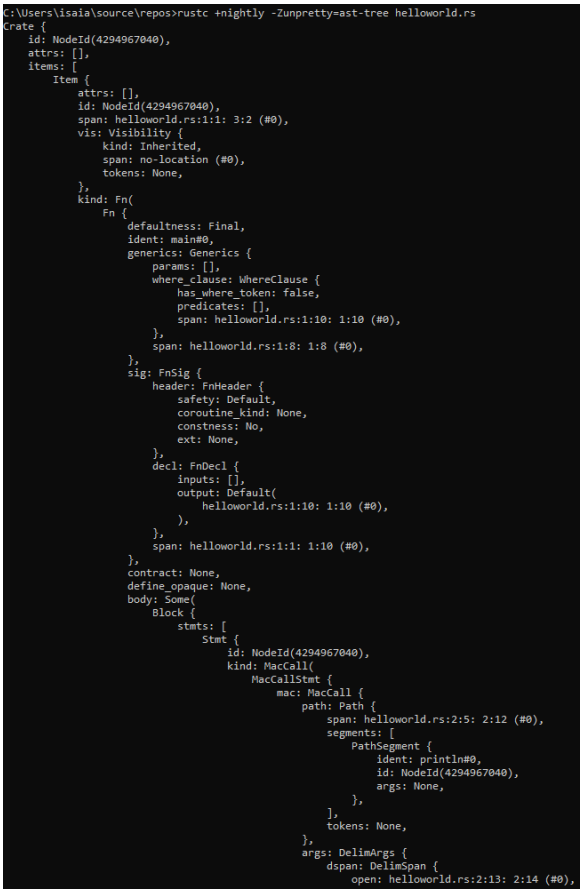


Figure 15: AST Tree.

After that, the compilation process produces the high-level intermediate representation.

```
#[attr = MacroUse {arguments: UseAll}]
extern crate std;
#[prelude_import]
use ::std::prelude::rust_2015::*;
fn main() {
    { ::std::io::_print(format_arguments::new_const(&["Hello, world!\n"])); };
}
```

Figure 16: HIR.

The last intermediate representation that the -Z unpretty command lets us see is the MIR, which is basically a control flow graph.

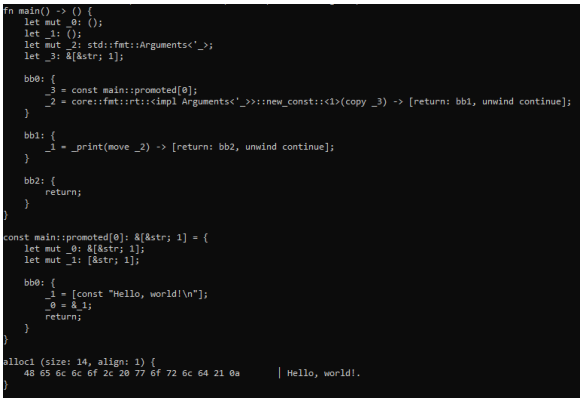


Figure 17: MIR.

The compilation ends with an executable file that allows us to run our code.

Figure 18: Executable hello world.

4.2 Interpreters

Here we can see a simple code snippet exemplifying the input given to the interpreter, in this case JavaScript. The classic "Hello World" program is used as an example, where the interpreter receives a **.js** file as input and the output is displayed in the terminal.

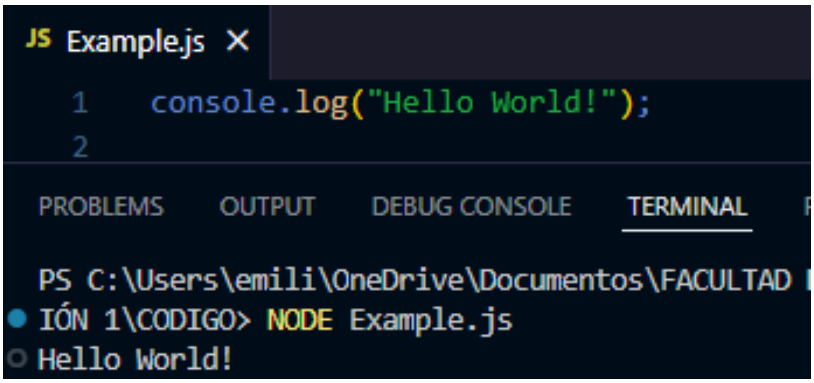


Figure 19: Example of execution JavaScript interpreter

Next, by using the command node –**print-bytecode Example.js**, we can visualize the intermediate code generated by the V8 engine for the JavaScript program. As we discussed earlier the bytecode represents the translation of the source code into a lower-level form that the interpreter can execute efficiently, including the instructions, registers, and constant pool used during execution.

```
22529 S> 0000035B3C05412A @  162 : 17 1a             LdaImmutableCurrentContextSlot [26]
22539 E> 0000035B3C05412C @  164 : 31 04 2a          GetKeyedProperty a1, [42]
         0000035B3C05412F @  167 : c8                Star1
         0000035B3C054130 @  168 : 17 30             LdaImmutableCurrentContextSlot [48]
22548 E> 0000035B3C054132 @  170 : 43 f8 29          BitwiseAnd r1, [41]
         0000035B3C054135 @  173 : c8                Star1
         0000035B3C054136 @  174 : 0c                LdaZero
22559 E> 0000035B3C054137 @  175 : 70 f8 2c          TestEqualStrict r1, [44]
         0000035B3C05413A @  178 : 9d 12             JumpIfTrue [18] (0000035B3C05414C @ 196)
22573 S> 0000035B3C05413C @  180 : 17 52             LdaImmutableCurrentContextSlot [82]
         0000035B3C05413E @  182 : c8                Star1
         0000035B3C05413F @  183 : 11                LdaTrue
         0000035B3C054140 @  184 : c5                Star4
         0000035B3C054141 @  185 : 19 03 f7          Mov a0, r2
         0000035B3C054144 @  188 : 19 04 f6          Mov a1, r3
22573 E> 0000035B3C054147 @  191 : 63 f8 f7 03 2d    CallUndefinedReceiver r1, r2-r4, [45]
         0000035B3C05414C @  196 : 0e                LdaUndefined
22613 S> 0000035B3C05414D @  197 : ae                Return
Constant pool (size = 9)
0000035B3C053FF1: [TrustedFixedArray]
 - map: 0x025196a809a9 <Map(TRUSTED_FIXED_ARRAY_TYPE)>
 - length: 9
           0: 0x00b4dd518749 <String[6]: #kState>
           1: 0x012134618f01 <String[18]: #kAfterWritePending>
           2: 0x012134618d39 <String[7]: #kEnding>
           3: 0x012134618d19 <String[10]: #kNeedDrain>
           4: 0x00b4dd534d61 <String[10]: #kDestroyed>
           5: 0x025196a817c9 <String[6]: #length>
           6: 0x00b4dd50cce1 <String[4]: #emit>
           7: 0x012134634b19 <String[5]: #drain>
           8: 0x012134635fb9 <String[9]: #pendingcb>
Handler Table (size = 0)
Source Position Table (size = 97)
0x035b3c054151 <Other heap object (TRUSTED_BYTE_ARRAY_TYPE)>
```

Figure 20: Intermediate code generated by V8 engine from the JavaScript program.

For Python, we will use the `dis` module to get the bytecode for a function that prints `"Hello, world!"` and another that gives the square of any positive number.

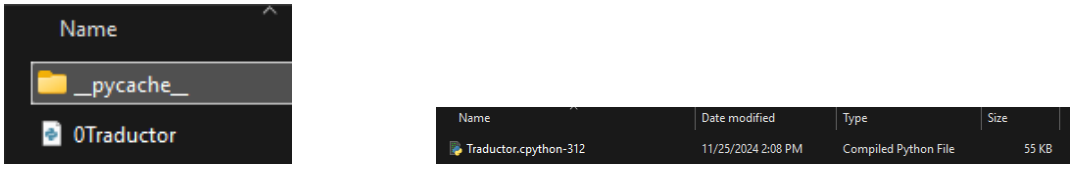
```
1   import dis
2
3   def main():
4       print("Hello, world!")
5
6   print("- Bytecode Hello World -")
7   dis.dis(main)
8
9   def square(x):
10      if x > 0:
11          return x * x
12      else:
13          return 0
14
15  print("- Bytecode Function -")
16  dis.dis(square)
17
```

```
PROBLEMS 6   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   Python: prueb

PS C:\Users\gmene\Documents\Python\extra> & C:/Users/gmene/AppData/Loca
/Python312/python.exe c:/Users/gmene/Documents/Python/extra/prueba.py
- Bytecode Hello World -
  3           0 RESUME                   0

  4           2 LOAD_GLOBAL              1 (NULL + print)
             12 LOAD_CONST               1 ('Hello, world!')
             14 CALL                     1
             22 POP_TOP
             24 RETURN_CONST             0 (None)
- Bytecode Function -
  9           0 RESUME                   0

 10           2 LOAD_FAST                0 (x)
              4 LOAD_CONST               1 (0)
              6 COMPARE_OP              68 (>)
             10 POP_JUMP_IF_FALSE        5 (to 22)

 11          12 LOAD_FAST                0 (x)
             14 LOAD_FAST                0 (x)
             16 BINARY_OP                5 (*)
             20 RETURN_VALUE

 13     >>   22 RETURN_CONST             1 (0)
```

Figure 21: Enter Caption

The bytecode is read as follows: the first column has the line numbers of the original source code file. The second column has >> that indicates the destination of a JUMP. The third column is the operation address with the operation name. The fourth column contains the operation parameters and the last column contains annotations to help correlate the original file with the bytecode [37].

In order to generate a `__pycache__` and a file inside with termination `cpython-312.pyc` a module has to be imported, if it is executed in the main script it does not generate, but another example for a robust project could look like shown in Figure 22.



(a) `__pycache__` folder. (b) `.pyc` file generated.

Figure 22: Intermediate code file generation for an example project.

4.3 Assemblers

First we wrote a hello world program using both GAS and NASM syntax:

```
 GNU nano 6.2          gas.s
.data
hello:
    .ascii "Hello, World!\0 "
.text
.global _start

_start:
    mov $1, %rax # syscall: sys_write
    mov $1, %rdi # file descriptor: stdout
    mov $hello, %rsi # string address
    mov $13, %rdx # string length
    syscall # calls the kernel

    mov $60, %rax # syscall: sys_exit
    xor %rdi, %rdi # exit status: 0
    syscall # calls the kernel
```

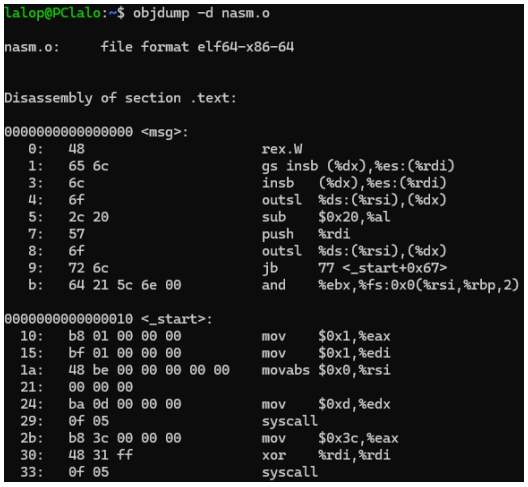
(a) GAS source file

```
 GNU nano 6.2          nasm.s
section.data
    msg db 'Hello, World!', 0

section.text
    global _start
_start:
    mov rax, 1 ; syscall: write
    mov rdi, 1 ; file descriptor: stdout
    mov rsi, msg ; pointer to message
    mov rdx, 13 ; message length
    syscall; make the syscall
    mov rax, 60 ; syscall: exit
    xor rdi, rdi ; status: 0
    syscall; make the syscall
```

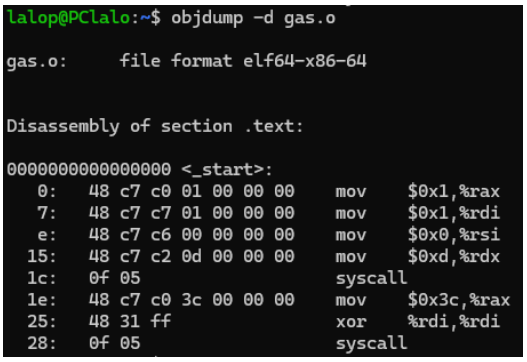
(b) NASM source file

Figure 23: Source file input, original code by [43]

Now we use both programs ('NASM' and 'as') to assemble the source file into an object file and compare the .o output of both programs:



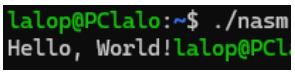
(a) NASM object file contents



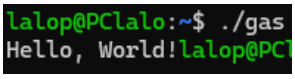
(b) GAS object file contents

Figure 24: objdump of NASM and GAS .o files

As we can see both assemblers produce a very similar output (ignoring the string not being shown in the GAS object file), when we link the programs we get the following output:



(a) NASM object file contents



(b) GAS object file contents

Figure 25: Terminal output of GAS and NASM programs

5 Conclusions

Language processors come in various forms, yet, every single one strives to follow the principles of correctness with which it was designed, to ensure that it sticks to the designated grammar rules and syntax in every phase of the input processing in order to get satisfying results. Each language got created with its own needs in mind, and we can see it in the different levels of abstraction, usability, portability and efficiency they posses for different scenarios. This knowledge will help us recognize which tool could work better for which case and give us ample knowledge to manipulate them having the structure in mind and not shying away from interacting with low-level language, even if it is only for debugging or optimizing computer programs.

The compilation process of C language forms the basis of many commonly used languages. In this case, it helped us to understand concepts and to apply them in other translators or programming languages. The examples included in the results allowed us to clearly see the toolchain of the compilation, so the generation of intermediate files is the proof of different optimized stages.

The study of interpreters and the example of JavaScript with the V8 engine and Python with a compilation step demonstrate how modern programming languages can combine them to balance efficiency and flexibility. Highlighting the advantages and disadvantages of a direct execution with its counterparts in terms of run-time, error management, etc. It is one of the main reasons why interpreters play a crucial role in execution programming languages by reading source code, parsing it according to formal grammar rules, and converting it into actions or intermediate representations at run-time.

Our research into assemblers has given us a deeper understanding of the way assemblers work, the available standards including the pros and cons of the two examples used for this assignment, as well as a broader understanding of the back-end functions of the GCC compiler that we have previously relied on for our programming assignments.

Although this research only allowed us to establish a general overview of the present course, it enabled us to accomplish the main objectives, which were to gain a preliminary understanding of how the programming languages we commonly use are structured in their grammar and also how they function in their own but similar way from human language to formal language to a machine. This also provides a comparison of the behavior of compilers, interpreters, and assemblers identifying how they respond to different needs, in order to decide which tool could help us more like we said before. Finally, another objective proposed by this task was to observe the work dynamics, so we were able to conclude that this indirect objective was achieved thanks to a respectful and fluid communication, a reasonable distribution of work, and the disposition of the working team.

References

[1] P. Calingaert, *Assemblers, Compilers and Program Translation*. Computer Science Press, 1979, ISBN: 0-914894-23-4.

[2] Universidad Nacional Autónoma de México, Facultad de Ingeniería, *Programa de Estudio: Compiladores*, Ciudad de México, México, 2005. [Online]. Available: %7Bhttps://www.ingenieria.unam.mx/programas_academicos/licenciatura/Computacion/07/compiladores.pdf%7D.

[3] 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.

[4] B. J. Gough, *An Introduction to GCC: for the GNU Compilers gcc and g++*. Network Theory Ltd, 2004, Foreword by Richard M. Stallman, Paperback (6"x9"), 124 pages. RRP £12.95 ($19.95), ISBN: 0954161793. [Online]. Available: https://web.archive.org/web/20051211093652/http://www.network-theory.co.uk/docs/gccintro/gccintro_8.html.

[5] 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.

[6] Y. Srikant and P. Shanka, *The Compiler Design handbook: Optimizations and Machine Code Generation*. CRC Press, 2003, ISBN: 0-8493-1240-X.

[7] C. W. Fraser and D. R. Hanson, *A Retargetable C Compiler: Design and Implementation*. Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1984.

[8] B. W. Kernighan and D. M. Ritchie, *The C Programming Language* (Prentice Hall Software Series), 2nd. Prentice Hall, 1988, ISBN: 978-0131103627.

[9] *Programming languages — c*, International Standard, Committee Draft N1570, April 12, 2011, International Organization for Standardization and International Electrotechnical Commission, 2011.

[10] GNU Project. "Gimple representation — gnu compiler collection (gcc) internals." Accedido: 21-ago-2025. (2010), [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html.

[11] GNU Project. "Ssa representation — gnu compiler collection (gcc) internals." Accedido: 21-ago-2025. (2010), [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/SSA.html.

[12] GNU Project. "Rtl representation — gnu compiler collection (gcc) internals." Accedido: 21-ago-2025. (2010), [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/RTL.html.

[13] P. H. Dave and H. B. Dave, *Compilers: Principles and Practice*, English. Pearson India, 2012, p. 504, Intermediate to Advanced.

[14] Rust Project Developers, *Getting started*, https://rustc-dev-guide.rust-lang.org/getting-started.html, [Online; accessed 21-Aug-2025], 2025.

[15] N. Matsakis, *Introducing mir*, Accessed: 2025-08-21, 2016. [Online]. Available: https://blog.rust-lang.org/2016/04/19/MIR/.

[16] The Rust Programming Language Team, *Notation*, https://doc.rust-lang.org/reference/notation.html, Accedido: 2025-08-21, 2025.

[17] C. Minnick, *JavaScript All-in-One For Dummies*. Wiley, 2023, ISBN: 9781119906834.

[18] A. Boduch, *JavaScript Concurrency*. Packt Publishing, 2015, ISBN: 978-1-78588-923-3.

[19] E. International. "Ecmascript® 2023 language specification (14th edition)," Ecma International. (Jun. 2023), [Online]. Available: https://262.ecma-international.org/14.0/.

[20] V. Team, *Firing up the Ignition interpreter*. V8 Developers, Aug. 2016. [Online]. Available: https://v8.dev/blog/ignition-interpreter.

[21] V. Team, *Background compilation*. V8 Developers, Mar. 2018. [Online]. Available: https://v8.dev/blog/background-compilation.

[22] C. Horstmann and R. Necaise, *Python for Everyone*, 2nd. John Wiley Sons, 2016, ISBN: 978-1-119-05655-3.

[23] N. Nishchhal, *Python Made Easy*. Notion Press, 2020, ISBN: 78-1-64983-726-4.

[24] P. S. Foundation. "The python language reference: Top-level components." version 3.13.7. (Aug. 19, 2025), [Online]. Available: https://docs.python.org/3/reference/toplevel_components.html.

[25] Python Software Foundation, *Exec and eval functions*, https://docs.python.org/3/library/functions.html, Accessed: August 20, 2025, 2025.

[26] G. M Sirish Kumar and A. Reddy, *PYTHON PROGRAMMING*. RAJA SURESH, 2023, ISBN: 9788196176938.

[27] B. K. M. Hailperin and K. Knight, *Concrete Abstractions: An Introduction to Computer Science Using Scheme*. Brooks/Cole Publishing Company, 1999, ISBN: 9780534952112.

[28] G. van Rossum, P. Galindo, and L. Nikolaou. "Pep 617 – new peg parser for cpython." PEP 617 – Final, Standards Track. Last modified 2025-02-01, Python Software Foundation. (Mar. 24, 2020), [Online]. Available: https://peps.python.org/pep-0617/.

[29] C. N. Fischer. "Ll(1) grammars." CS-536 Course., University of Wisconsin-Madison. (Oct. 25, 2012), [Online]. Available: https://pages.cs.wisc.edu/~fischer/cs536.f12/lectures/Lecture22.pdf.

[30] P. S. Foundation. "Python language reference: Full grammar specification." version 3.13.7. (Aug. 19, 2025), [Online]. Available: https://docs.python.org/3/reference/grammar.html.

[31] Python Software Foundation, *Lexical analysis*, https://docs.python.org/3/reference/lexical_analysis.html, The Python Language Reference, accessed August 21, 2025, 2025.

[32] GNU Project, *Peg syntax reference*, https://www.gnu.org/software/guile/manual/html_node/PEG-Syntax-Reference.html, Accessed: August 20, 2025, GNU Project, 2025.

[33] A. Sweigart, *Beyond the Basic Stuff with Python: Best Practices for Writing Clean Code*. No Starch Press, 2020, ISBN: 9781593279660.

[34] J. McDonald, *Dead Simple Python. Idiomatic Python for the Impatient Programmer*. No Starch Press, 2022, ISBN: 9781718500938.

[35] Python Software Foundation, *Bytecode*, https://docs.python.org/es/3/glossary.html#term-bytecode, Python Documentation Glossary. Accedido: 21 de agosto de 2025, 2025.

[36] G. Lanaro, *Python High Performance*. Packt Publishing, 2017, ISBN: 9781787282438.

[37] M. Gorelick and I. Ozsvald, *High Performance Python*. O'Reilly Media, 2014, ISBN: 9781449361778.

[38] [Online]. Available: https://students.mimuw.edu.pl/~zbyszek/asm/arm/assembler-intro.pdf.

[39] D. Elsner and J. Fenlason, *Using as the gnu assembler january 1994*, 1994. [Online]. Available: https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_node/as_toc.html.

[40] The NASM Development Team, *NASM — The Netwide Assembler*, 2003. [Online]. Available: https://userpages.cs.umbc.edu/chang/cs313.f04/nasmdoc/nasmdoc.pdf.

[41] R. Narayan, *Linux assemblers: A comparison of GAS and NASM*, en, 2007. [Online]. Available: https://developer.ibm.com/articles/l-gas-nasm/ (visited on 08/21/2025).

[42] H. A. Valdecantos, "Una introducción al compilador c de gnu," *Ciencias Exactas y Tecnología (cet)*, no. 37, 2010, ISSN: 1668-9178.

[43] M. Oliveira, *X86_64 Assembly Tutorial with GNU Assembler (GAS) for Beginners*, 2024. [Online]. Available: https://terminalroot.com/x8664-assembly-tutorial-with-gnu-assembler-gas-for-beginners/ (visited on 08/21/2025).