Universidad Nacional Autónoma de México
Facultad de Ingeniería

Computer Engineering

Subject: Compilers (0434)

Account numbers:
320075669
423066940
423035975
320199866
320061682

Group: 5

Semester: 2026 - I

Mexico, CDMX. August 19th 2025

Contents

Compilers
Group: 5
UNAM / FI / DIE
Semestre 2026-1

Assigment 1

1 Introduction

- **Problem Statement**: Research examples of compilers, interpreters, and assemblers, in which you will be requiered to identify:
  - What input does it requiere?
  - The main elements of its grammar.
  - The intermediate code it generates.
- **Reason**: This homework is important because it will allow us to understand the main elements of the grammar, its inputs, and the code it can generate as an intermediary.
- **Objectives**: This assignment will allow us to identify the focus of the research, leading to a better understanding of the topic.

2 Theoretical Framework

Compiler Theory

A compiler is a translator program that converts source code written in a high-level programming language like C, C++, Python, Java, etc., into target machine code, following the professor's explanation in class. The compilation process is organized into multiple phases including lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and code generation.

Interpreter Theory

An interpreter is a program that allows processing the source code of a software project during its runtime, that is, it is executed line by line, so it reads, analyzes, and prepares each sequence consecutively for the processor. While the software is running, it acts as an interface between the project and the processor. Something very important about the interpreter is that it does not create an executable file.

Assembler Theory

An assembler is a program that translates code in assembly language into machine code, which allows direct communication with computer's hardware. Assemblers are used for low-level programming and are specific to a particular computer architecture.

3 Body

3.1 Compilers

GNU Compiler Collection (GCC)

GCC is one of the most sophisticated and commonly used compiler systems, supporting multiple programming languages and target architectures.

- **Input Requirements:** GCC requires as input source code files written in supported high-level languages like C, C++ or Fortran. Those files contain programming constructs including variable declarations, function definitions and control flow statements.
- **Grammatical Elements:** The grammar for languages handled by GCC conforms to Context-Free Grammar (CFG) specifications.
  - Lexical tokens: identifiers, keywords, operators, constants, delimiters
  - Syntax rules: expression structures, statement formations, declaration syntax
  - Semantic rules: type compatibility, scope rules, function prototypes

```c
#include <stdio.h> //Preprocessor directive

int main() // Funciont definition
{
    int x = 10; // Variable declaration

    if (x > 5 ){ // Control statement
        printf("Value: %d \n", x);  // Function call
    }

    return 0; // Return statement
}
```

Clang Compiler

The **Clang compiler** requires as input a *source code file* written in a supported language (C, C++, Objective-C, Objective-C++, etc.).

- For **C**: files with the `.c` extension
- For **C++**: files with extensions such as `.cpp`, `.cc`, `.cxx`
- For **Objective-C**: files with the `.m` extension
- For **Objective-C++**: files with the `.mm` extension

3.1.1 Main Grammar Elements:

- Expressions: arithmetic operations, function calls.
- Declarations: variables, functions, data types.
- Statements: control flow (`if`, `for`, `while`, `return`).
- Definitions: function and struct implementations.
- Translation unit: the entire file processed with includes and macros.

3.1.2 Intermediate Code (LLVM IR):

Generated from the source code before final compilation. As an example:

```
@.str = private constant [12 x i8] c"Hello world\0A\00"

define i32 @main() {
```

```
entry:
  %0 = call i32 (i8*, ...) @printf(i8* getelementptr
     ([12 x i8], [12 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}
```

3.2 Assemblers

Netwide Assembler

Netwide Assembles, commonly known as NASM, is an open-source assembler for the x86 architecture that supports a variety of output formats, like binary, ELF, COFF, and others.

- **Input Requirements:** The Netwide Assembler, primarily requires a source file, written in assembly language, as its primary input. This file, usually with an .asm extension, contains the assembly instructions written in NASM's syntax.
- **Grammatical Elements:** The expressions in NASM are similar in syntax to those in C.
  - Layout of a NASM Sorce Line: Each NASM source line contains some combination of the four fields
    `label: instruction operands ; comment`
  - Pseudo-Instructions: The current pseudo-instructions are `DB, DW, DD, DQ and DT`, also, their uninitialised counterparts `RESB, RESW, RESD, RESQ and REST`, additionally, the `INCBIN` command, that includes External Binary Files, and `EQU` command, that defines constants, and the `TIMES` prefix, that repeats instructions or data.
  - Effective Addresses: It is any operand to an instruction wich reference memory. They consist of an expression evaluating to the desired address, enclosed in square brackets. Constants: There are four different types of constants: numeric, of which can be hex, octal and binary; character, can be up to four characters between single or double quotes; string, these looks like a character constant, only longer, also, these are only acceptable to some pseudo-instructions, namely the `DB` family and `INCBIN`; and floating-point, these are acceptable only as arguments to `DD, DQ and DT`.
  - Expressions: These are similar in syntax to those in C. Among which the following stand out: Bitwise OR Operator (|), Bitwise XOR Operator (^), Bitwise AND Operator (&), Bit Shift Operators (<< and >>), Addition and Subtraction Operators (+ and −), Multiplication and Division (∗, /, //, % and %%), Unary Operators (+, −, ∼ and `SEG`).
- **The intermediate code it generates:** Unlike high-level compilers such as C or Java, which generate intermediate machine-independent code, assemblers such as NASM do not produce an abstract Intermediate Representation (IR); Instead, they directly translate assembler instructions into machine code, packaged within an .o object file.

Microsoft Macro Assembler

Microsoft Macro Assembler (MASM), is an assembler for the x86 architecture that translates assembly language source code into machine code.

- **Input Requirements:** MASM uses Intel's standard instruction format, with operands that work with registers, memory addresses, directives, etc, using their proper prefixes. MASM use a single-pass assembly, which means that code must be written so that all definitions are available when referenced. Additionally, MASM requires setting the correct CPU segmentation mode.

- **Grammatical Elements:** Like many other compilers, MASM primarily uses reserved words, identifiers, and constants or literals. Below we explain the main grammatical elements:
  - Reserved words: These are predefined keywords in MASM that represent instruction mnemonics, operators, and directives, such as `MOV`, `ADD`, `.CODE`, `TYPE`, etc.
  - Identifiers: These are names defined by the user for variables, procedures, labels, macros, etc.
  - Constants: These can be numeric, string, or character values. This category also includes literals.

  Together, the elements above are used to create expressions, which are combinations of identifiers, constants, and operators.
- **The intermediate code it generates:** Like NASM, MASM does not produce an abstract IR, it directly translates assembler instructions into machine code, generating an .obj object file.

3.3 Interpreters

Python

Python is a powerful and easy-to-learn programming language. It features efficient high-level data structures and a simple yet effective object-oriented programming system. Python's elegant syntax and dynamic typing, along with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas, across most platforms. Python is not compiled to native machine code, instead the process is: parsing, then compilation and finally execution

- **Input Requirements:**

  Python requires source code written in plain text that follows its syntax rules. Typically stored in a file with a .py extension. The input can also be provided interactively through the Read-Eval-Print Loop. The input is readable for humans, consisting of statements, expressions and definitions, just like classes, functions, etc.
- **Grammatical Elements:**
  - Single input: is a single interactive statement.
  - File input: is a module or sequence of commands read from an input file.
  - Eval input: is the input for the *eval()* functions.
  - Func type input: is a PEP 484 Python 2 function type comment.
  - Arguments: parameters and set of rules used (like arguments, kwargs, varargslist, etc.).

Ruby

Ruby is a high-level, interpreted, general-purpose programming language, created in 1995 by Yukihiro "Matz" Matsumoto in Japan. Ruby code is not compiled directly to machine code; instead, it runs inside an interpreter (MRI/YARV). Everything in Ruby is an object — even numbers, strings, and nil. You don't declare variable types; they are determined at runtime.

- **Input Requirements:** The Ruby interpreter (MRI) requires Ruby source files (.rb) as input. The source is plain text written in the Ruby language, which is defined by the Ruby language specification (though less formal than Java's JLS). Ruby programs usually contain:
  - Classes and modules

  - Methods and blocks
  - Variables and constants
  - Control structures (if, while, for, case, etc.)
  - Expressions and operators

- **Grammatical Elements:** Ruby's grammar can be divided into two levels: This level explains how the source code text is broken into tokens.

  - Identifiers: Names for variables, constants, methods, classes, and modules.
  - Keywords: Cannot be used as identifiers. Examples are: class, def, if, else, elsif, end, while, for, return, yield, super, true, false, nil.
  - Literals: Constants that represent fixed values. They can be numeric, boolean or null.
  - Operators:
    Arithmetic: +, -, *, /,
    Comparison: ==, !=, ¡, ¿, ¡=, ¿=
    Assignment: =, +=, -=, *=
  - Separators and punctuation: They can be parentheses, brackets, braces, Comma, semicolon, dot, etc.

4 Conclusions

After completing the points of the assignment, we can conclude that we have reached the objectives established at the beginning because we, as a team, realized about all the components that conform the topics we are going to see, such as compilers, interpreters and assemblers. Knowing about the elements of their grammar and the intermediate code they generate let us understand how they work and what happens in the background. This could help us to optimize or simplify some tasks or points bout their functionality.

References

[1] Lenovo. (s. f.). *¿Qué es un ensamblador?*. Retrieved from https://www.lenovo.com/mx/es/glosario/ensamblador/

[2] LLVM Project. (s. f.). *The LLVM Compiler Infrastructure*. Retrieved from https://llvm.org/

[3] Viking Software. (s. f.). *What is Clang?*. Retrieved from https://www.vikingsoftware.com/services/technologies/what-is-clang/

[4] Programación Pro. (s. f.). *¿Qué es un compilador? Ejemplos y explicación detallada*. Retrieved from https://programacionpro.com/que-es-un-compilador-ejemplos-y-explicacion-detallada/

[5] IONOS. (s. f.). *Compilador e intérprete: diferencias y ejemplos*. Retrieved from https://www.ionos.mx/digitalguide/paginas-web/desarrollo-web/compilador-e-interprete/

[6] IMMUNE Technology Institute. (s. f.). *¿Qué es un compilador?*. Retrieved from https://immune.institute/blog/que-es-un-compilador/

[7] Python Software Foundation. (s. f.). *Tutorial de Python 3*. Retrieved from https://docs.python.org/es/3/tutorial/

[8] Python Software Foundation. (s. f.). *Referencia de gramática: Python 3.8*. Retrieved from https://docs.python.org/es/3.8/reference/grammar.html

[9] Microsoft. (s. f.). *MS MASM 6.0 Programmer's Guide*. In *Microsoft Programmer's Library 1.3*. Retrieved from https://www.pcjs.org/documents/books/mspl13/masm/mpguide/

[10] Matsumoto, Y., & Ruby Core Team. (2023). *Ruby programming language documentation.* Ruby Official Website. Retrieved from https://www.ruby-lang.org/en/documentation/

[11] UMBC. (s. f.). *NASM — The Netwide Assembler*. University Of Maryland, Baltimore County. Retrieved from https://userpages.cs.umbc.edu/chang/cs313.f04/nasmdoc/nasmdoc.pdf

[12] Howdy. (s. f.). *Netwide Assembler.* Retrieved from https://www.howdy.com/glossary/netwide-assembler

[13] Howdy. (s. f.). *Microsoft Macro Assembler.* Retrieved from https://www.howdy.com/glossary/microsoft-macro-assembler