COSC 4780 - Principles of Programming Languages - Spring 2022

Lecture 0: Course Introduction

Finley McIlwaine

University of Wyoming

fmcilwai@uwyo.edu

January 18, 2022

About the instructor

- Fin, Finley, Mr. McIlwaine, whatever you're comfortable calling me
- B.S. in Computer Science from University of Wyoming (2020)
- Currently a graduate student in the WABL lab
- Office hours T,TR 9-10AM on Zoom (look at syllabus)
- Email me anytime at all with questions, concerns, thoughts, etc.:

fmcilwai@uwyo.edu

Grader: Andey Robins (jtuttle5@uwyo.edu).

Office hours M,W 9-11AM EERB 228

Let's talk about where you're at...

Goals for This Course

After taking this course, you will:

- Have a foundational understanding of the design and analysis of programming languages
- Know how to build interpreters for programming languages
- Understand the phases of compilation, bridging the gap between high-level code and machine languages
- **Be able to recognize and solve compiler problems in the wild**

. . . and much more!

Computers, Compilers, Interpreters

At a very high level, and in the most common conceptualization, computers are machines that read and manipulate binary data.

Sequences of 0s and 1s are enough to describe any object, and can encode many things. For example: Programs and data.

Representing Data

We know that numbers are easily represented in binary. For example, the integers:

```
0 = 0
1 = 1
2 = 10
3 = 11
4 = 100
```

In the case of numbers, the mapping from binary to base-10 is algorithmic. We are staying within a single type and simply changing the representation of values.

Representing Data

Representing types outside of numbers requires us to impose structure. For example, characters and ASCII encodings:

```
A = 65 = 10000001
B = 66 = 10000010
C = 67 = 10000011
```

These numbers represent those characters only because we impose the ASCII encoding of characters. Enforcing other structures, and mapping numbers to different meanings is how we can begin to encode *programs*.

Representing Programs

Real-world example: The Java Virtual Machine (JVM) runs programs that are encoded as sequences of bytes, whose numeric values are mapped to semantically interesting *instructions* and *operations*. Integer addition and multiplication, for example:

```
+ =  96 = 0110 0000 = 60
* = 104 = 0110 1000 = 68
```

The 60 and 68 are the hexadecimal encodings. You can find these values for yourself on WIKIPEDIA.

iadd	60	0110 0000		value1, value2 → result	add two ints
imul	68	0110 1000		value1, value2 → result	multiply two integers

Representing Programs

The JVM is a stack-based machine, much like the one from 3015 last semester! In order to operate on numbers, we must have their values on the stack. Enter instruction `0x10`:

bipush	10	0001 0000	1: byte	→ value	push a *byte* onto the stack as an integer *value*

The instruction `bipush 2` is therefore encoded in binary as `0001 0000  0000 0010`.

Representing Programs

We can now encode interesting arithmetic expressions in binary. For example $5 + 6 * 7$ is computed in the JVM by the following program:

```
bipush 5
bipush 6
bipush 7
imul
iadd
```

$\Rightarrow$

```
0001 0000  0000 0101
0001 0000  0000 0110
0001 0000  0000 0111
0110 1000  0110 0000
```

The program on the left is shown as *assembly code*, which is the human-readable symbolic version of machine code.

Representing Programs

Of course, for more interesting and useful programming constructs we will need more instructions, but the important takeaway is:

- **Both data and programs can be represented as binary code**.
- **There is a systematic translation from conventional expressions to binary code**

That translation from some sort of source language to some sort of target language (here numeric expressions and JVM bytecode) is exactly the job of a *compiler*.

Compilation

The two main ideas of compilation:

- **Syntactic analysis:** Analyse expressions into an operator `F` and operands `X` and `Y`.
- **Syntax-directed translation:** Compile the code for `X` followed by the code for `Y`, followed by the code for `F`.

Both of these ideas are recursive. Think how you might compile the following:

```
5 * ((6 - 7) * 8)
```

Levels of Languages

The task of a compiler may be more or less demanding depending on the distance between the source and target languages–much like translation between spoken languages.

This concept of "distance" is clearer when examining programming languages, as we usually do it with respect to machine language as a common reference point. One ordering of languages in increasing distance from machine language is shown below:

machine language $\rightarrow$ assembly $\rightarrow$ C $\rightarrow$ C++ $\rightarrow$ Java $\rightarrow$ Haskell $\rightarrow$ human

Compilation and Interpretation

Compilers *translate* a source representation of a program into some target representation. They do not actually run the program. Interpreters do that. For example, a source language expression:

$$5 + 6 * 7$$

is by an interpreter turned into its value: 47

Interpretation can be done without any translation of the source code by simply traversing an abstract representation of the source program itself. However, compilation followed by interpretation is a common approach (JVM).

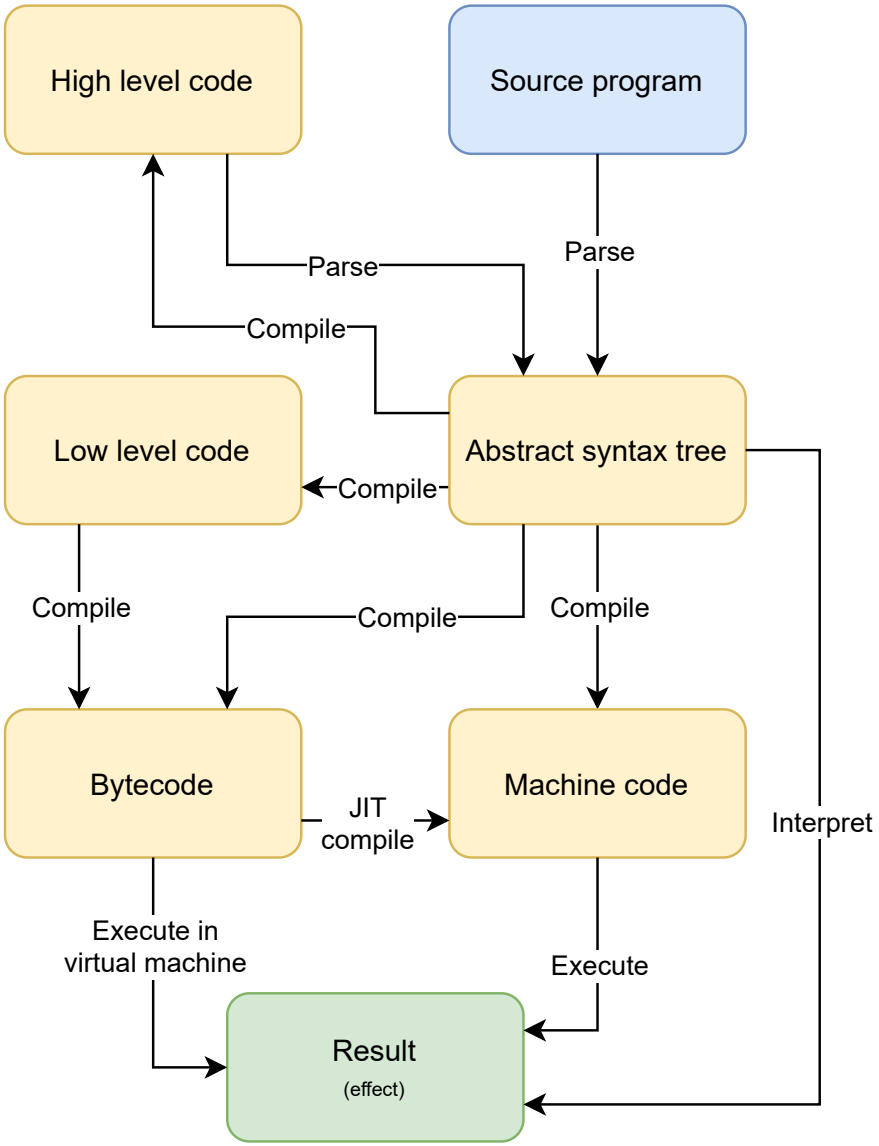
Compilation and Interpretation

As you might expect, there are different advantages to interpretation and compilation:

Interpretation	**Compilation**
Easier to implement	Machine code is faster to execute
More easily portable	"Lower" targets faster to interpret
Easier to	More opportunity for optimization

The line between interpretation is blurred by the things like just-in-time (JIT) compilation and virtual machines with real machine language instruction sets.

Compilation and Interpretation



Compilation Phases

Obviously, there are many flavors of the general ideas of compilation and interpretation. Different languages with different goals are structured/compiled/interpreted in drastically different ways. However, one thing that is true for *most* general-purpose language implementations is that compilation and interpretation is broken up into discrete and composable phases.

Each phase exists to either translate, analyse, or interpret the result of previous phases. The first phase's input is the raw source code as a string of characters.

Compilation Phases - Lexing

We call the first phase **lexing** or **tokenizing** or **scanning**. It chunks the source string into discrete **tokens** or meaningful words. These tokens are usually the smallest elements a language's syntax/grammar. For example, the source "5 * (6 - 3)" becomes:

```
[INT 5,STAR,LPAREN,INT 6,MINUS,INT 3,RPAREN].
```

Compilation Phases - Parsing

Next, we have parsing. In parsing, the tokenized input is processed into and abstract representation of the original program. In this stage, all concrete elements of the original program are erased and only the essential semantic elements remain. For example, the source program above may be coverted into the following abstract syntax tree:

```
MulExp (IntExp 5) (MinExp (IntExp 6) (IntExp 3)).
```

Compilation Phases - Type Checking

In type checking, the result of the parsing stage is typically a value that proves the syntactic-well-formedness of the source program. However, if our language is statically typed we must check that the types of all expressions are valid, either by checking explicit type annotations or inferring types of implicitly typed expressions. This may be automatically completed by a type checking algorithm.

Compilation Phases - Code Generation/Interpretation

Once we know the type of the program is correct, we can begin emitting the target code (compilation) or evaluating the input program (interpretation). There may be more phases, such as optimization, before (or after) code generation, but in the simplest sense, this is the final phase.

We will study each of these phases more, this is just a quick tour, and you don't need to worry about understanding all of this now.

How will we do this?

Homework #0!

The End