Compilers Are Interpreters

We use the words *compiler* and *interpreter* to hide (abstract) reality.

To understand what "is going on underneath", we need to examine how *CPU*s are constructed and what *apps* are.

CPU Hardware

CPUs (Central Processing Unit - a designation invented in the 1950's which is no longer true)[1] consist of a blob of asynchronous electronic semi-conductors made up of various oxides ("rust" is an oxide).

We abstract the blob of electronics via a low-level API.

The API is implemented in hardware, and exposes the hardware using ideas of:

- a set of registers (global, shared)
- memory (global, shared).

All of the software apps that we write today boil down to accesses to this API.

We *used to* access this API via a bank of switches that stored codes in *memory*.

Then, we invented *assembler* apps that could convert text into codes in memory.

Then, we invented *compiler* apps that could convert text into assembler and, ultimately, into codes in memory.

Then, we invented HLLs (High Level Languages) that further abstracted the

[1] Most apps are decentralized today (2021, internet), hence, the word *CPU* (Central Processing Unit) is not an accurate depiction of what is going on.

notion of poking codes into memory.

HLLs gave us more checking for common programming errors, like syntax checking and type checking.

Apps

All software *apps* (applications) boil down to a set of instructions in *machine code* that are interpreted by the underlying hardware.

The instructions are numeric codes (binary) that are poked into memory.

Memory is an array of words (bytes (int8), int16, … int64) indexed by a numeric index stored in the the PC (Program Counter) register.

The CPU performs a set of actions based on the binary code stored in memory indexed by the PC register. We use the word *interpret* for this behaviour.

Interpreters

Sometimes we write large apps that we call *interpreters.*

These apps usually step through some input data and perform actions based on the input data.

For example

```
…
if (data == "hello") { print ("hello"); }
if (data == "goodbye") {print ("goodbye"); }
…
```

Such apps - *interpreters* - are actually interpreted by the CPU hardware.

Running such apps many times incurs a speed penalty - the *interpreter* app needs to re-parse each string each time through and this, then, gets *interpreted* by the CPU hardware.

Parsing (understanding, interpreting) strings is an "expensive" operation, since the interpreter app doesn't know the length of the strings and must walk the strings from front-to-back each time through.

We *could* write smarter apps, for example we could compress the strings into some kind of binary code that could be interpreted by the CPU less expensively.

We use the word *compiler* to mean apps which do the above kind of compression.

Diagram

Compilers

Sometimes we write large apps that we call *compilers.*

Compilers are pre-processors.

Compilers take specifications and convert them into machine codes.

The specifications are usually in the form of *text* and follow certain rules the we call HLLs (High Level Languages).

Some versions of *compilers* generate machine codes directly, other versions generate *assembly* code that is further converted into machine codes (by other apps).

In the end, however the machine codes are generated, the result is *interpreted* by the CPU hardware.

Compilers are apps. Apps are interpreted by the CPU hardware.

Example of a Compiler

A ficticious example of compiling follows…

In the interpreter section we saw the action of a ficticious interpreter

```
…
if (data == "hello") { print ("hello"); }
if (data == "goodbye") {print ("goodbye"); }
…
```

How would you make this "better"?

- You could make it use less memory.
- You could make it run faster.

Let's look at making it run faster.

We could make the above code run faster by pre-processing it and using binary codes instead of strings.

For example, maybe we would rewrite the above as:

```
…
if (data == 0x50) { print ("hello"); }
if (data == 0x75) {print ("goodbye"); }
…
```

Here, `0x50` and `0x75` are "faster" to interpret because we don't need to walk (re-walk) the strings ("hello" and "goodbye").

What is the hidden *gotcha*, the hidden cost?

- We have to run an app (once) to pre-process the code (converting the strings into binary codes), and,
- we have to store the preprocessed result somewhere, and,
- every time we abstract something, we end up throwing information away, which then restricts our further thinking and ideas.

Yet, in the end, the net result is a speed-up. We pay the cost of pre-processing once, then we run the result (through the CPU) many times.

We do this preprocessing in a second app. We call this kind of app a *compiler*. This kind of app is actually an interpreter that gets interpreted by the CPU, but it saves us run-time in the end, for running the original app (now compressed).

There are lots of nuances in writing such pre-processors. A field of study has emerged, called *compiler writing*.

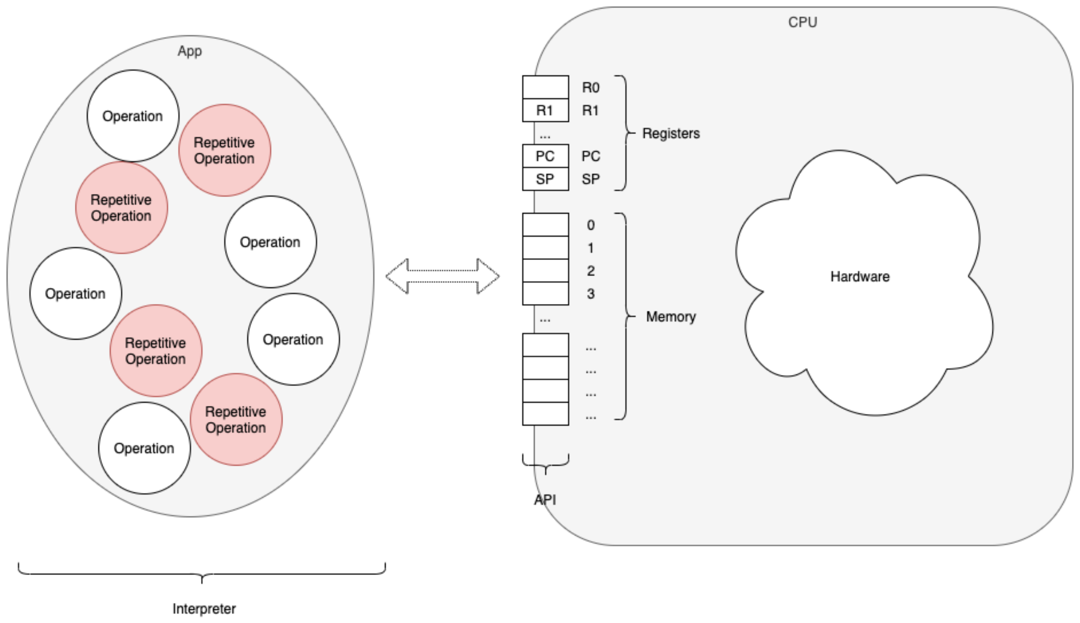
Language Design

It turns out that we can write better pre-processors if we put restrictions on the rules for writing the original app.

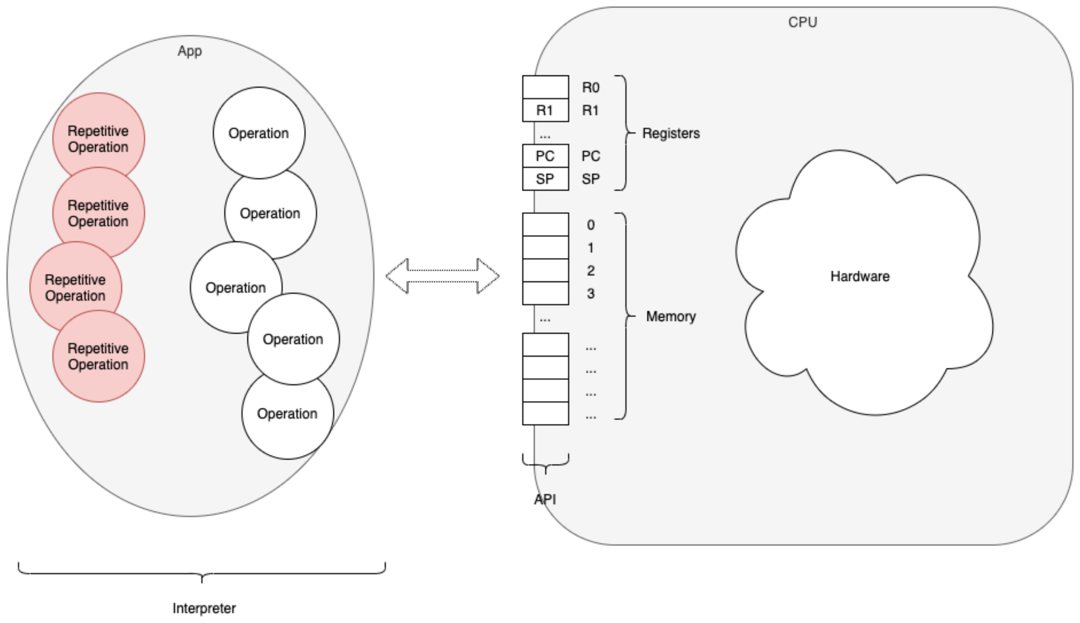
We call the activity of figuring out the rules, *language design*.

Diagrams

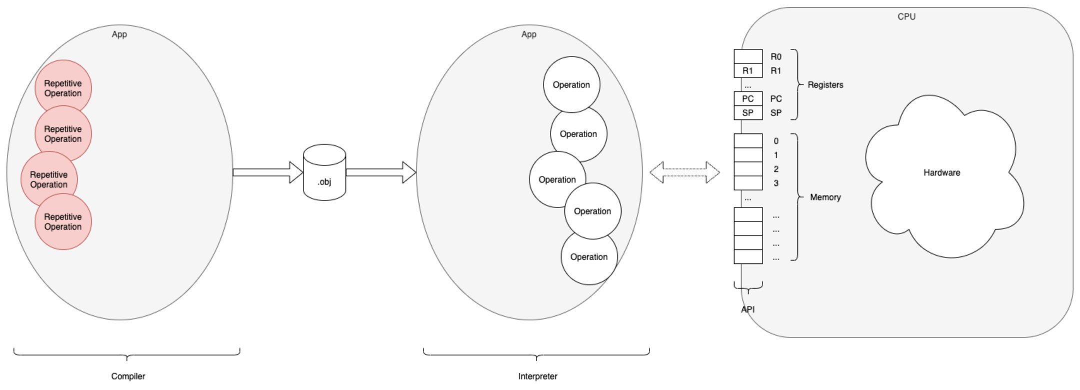
Interpreters



Refactoring Interpreters



Compilers and Interpreters


App
Repetitive Operation
Repetitive Operation
Repetitive Operation
Repetitive Operation
.obj
App
Operation
Operation
Operation
Operation
Operation
CPU
R1
...
PC
SP
R0
R1
PC
SP
Registers
0
1
2
3
...
...
...
...
...
Memory
API
Hardware
Compiler
Interpreter