Compiled & Interpreted Languages

*Programming Languages*

*William Killian*

*Millersville University*
Lecture Outline

• The Language Translation Pipeline
  • Scanning
  • Parsing
  • Analysis
  • Optimizing
  • Code Generation
• Running Your Program
• The Compiler
• The Interpreter
The Language Translation Pipeline
The BIG Picture

Source: Crafting Interpreters (https://craftinginterpreters.com)
Scanning

• A **scanner** takes in the linear stream of characters and chunks them together into **tokens**

\[
\text{var } \text{average} = (\text{min} + \text{max}) / 2;
\]

• Some characters don’t mean anything.
  • Whitespace is often insignificant
  • Comments, by definition, are ignored by the language.
  • The scanner usually discards these, leaving a clean sequence of meaningful tokens.

\[
\text{var } \text{average} \; = \; (\text{min} \; + \; \text{max}) \; / \; 2 \; ;
\]
Parsing

• A **parser** takes the sequence of tokens and builds a tree structure that mirrors the **grammar**.

```
var average = ( min + max ) / 2 ;
```

• Programming language experts call these tree structures “**syntax trees**”, “**ASTs**”, or just “**trees**”

![Diagram of a syntax tree with nodes labeled as follows:

- **Stmt.Var**: average
- **Expr.Binary**: /
- **Expr.Binary**: + 2
- **Expr.Variable**: min
- **Expr.Variable**: max
- **Expr.Literal**: 2
]
Analysis (i.e. Static Analysis)

• Binding / Resolution
  • For each **identifier** we find out where its **name** is **defined** and wire the two together.
  • This is where **scope** comes into play—the region of source code where a name refers to a declaration.

• Type Checking
  • Once we know where **names are defined**, we can also figure out their **types**. All operations must be valid.
  • If operations aren’t supported, we report a **type error**.
Optimizing

• Once we understand what the user’s program means, we can “change it”

• Optimizing is a **safe change to the program** that results in the **same semantics** (e.g. has the same behavior)

• The resulting program is usually **more efficient**

\[
pennyArea = 3.14159 \times (0.75 / 2) \times (0.75 / 2);
\]

\[
pennyArea = 0.4417860938;
\]

*Example*: Constant Folding
Code Generation

• The last step

• Converting it to a form the machine can run.
  • Usually **primitive assembly-like instructions** a CPU runs and not the kind of “source code” a human reads.

• Do we generate instructions for a real CPU or a virtual one?
  • **Real CPU**: Intel x86, ARM AArch64, IBM PowerPC, MIPS
  • **Virtual CPU**: LLVM IR, Microsoft CIL, Java Bytecode, Python bytecode
    • Advantages: Portable across Real CPUs
    • Disadvantages: Performance penalty
Running Your Program
Two Major Components

• Virtual Machine
  • Only needed for languages that target Virtual CPUs
  • Converts from Virtual CPU to Real CPU at runtime

• Language Runtime
  • Needed by all languages
  • The “extra pieces” required to have a language operate
Virtual Machine

- When you target a **Virtual CPU**, you need a program that converts the **Virtual CPU instructions** to **Real CPU instructions**

- This is done with a **Virtual Machine**
  - A program that emulates a hypothetical chip supporting your virtual architecture at runtime.
Language Runtime

- We usually need some services that our language provides while the program is running.

- Examples:
  - If a language automatically manages memory, we need a garbage collector running to reclaim memory.
  - If a language supports `instanceof`, then we need to keep track of the type of each object during execution.

- In a compiled language, the code implementing the runtime gets inserted into the resulting executable.

- If the language is run inside an interpreter or VM, then the runtime lives there.
Compilers and Interpreters
Compilers and Interpreters

• **BOTH** must perform all stages in the language translation pipeline
• The difference is *when* certain stages happen

• What languages do you think are compiled?
• What languages do you think are interpreted?
• Which languages do you think can be both?
## Compiled/Interpreted Languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Compiled</th>
<th>Both</th>
<th>Interpreted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C / C++</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCaml</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Javascript</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C#</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Common Languages

- javac
- GCC
- TypeScript
- CoffeeScript
- Rust
- clang
- C#
- Haskell
- CPython
- YARV (Ruby)
- Lua
- clox
- Go
- Guile (Scheme)
- Scala
- V8 (JS)
- PHP3
- PHP4
- MRI (Ruby)
- jlox
Compiled Languages

- **Compiling** is an *implementation technique* that involves translating a source language to some other form.
  - When you generate bytecode or machine code, you are compiling.
  - When you transpile to another high-level language you are compiling too.
- When we say a *language implementation* “is a *compiler*”, we mean it translates source code to some other form but doesn’t execute it.
Interpreted Languages

• When we say a language implementation is an interpreter, we mean it takes in source code and executes it immediately.
• It runs programs “from source”.
• There is no separate entity created
Language != Implementation

• Language Implementations can either be:
  • Compiled
  • Interpreted

• Notice how we didn’t mention a language?

• Statements:
  • I can write an interpreter for C++ (see cling)
  • I can write a compiler for Javascript
  • I can write an interpreter for C#
  • I can write a compiler for ____________
  • I can write an interpreter for ____________
Bonus Video

Interpreters and Compilers (Bits and Bytes, Episode 6)
https://www.youtube.com/watch?v=_C5AHaS1mOA