An **interpreter** is a program that executes another program.

- An interpreter implements a *virtual machine*, which may be different from the underlying hardware platform. Examples: Java Virtual Machine; VMs for high-level languages such as Scheme, Prolog, Icon, Smalltalk, Perl, Tcl.
- The virtual machine is often at a higher level of semantic abstraction than the native hardware. This can help portability, but affects performance.
### Interpretation vs. Compilation

**Source Program**

1. **Front End**
   - Intermediate representation (syntax tree, stack machine code, etc.)

2. **High-level Interpreter**
   - Examples: Java Virtual Machine

3. **Back End**
   - Assembly/machine code

4. **Low-level Interpreter**
   - Examples: SPIM (MIPS emulator)

5. **Hardware**

---

### Interpreter Operation

\( \text{ip} = \text{start of program}; \)

\[
\text{while } ( \neg \text{done } ) \{ \\
\quad \text{op} = \text{current operation at } \text{ip}; \\
\quad \text{execute code for } \text{op} \text{ on current operands}; \\
\quad \text{advance } \text{ip} \text{ to next operation}; \\
\}
\]
### Interpreter Design Issues

- **Encoding the operation**
  
  I.e., getting from the opcode to the code for that operation ("dispatch"):
  
  - byte code (e.g., JVM)
  - indirect threaded code
  - direct threaded code.

- **Representing operands**
  
  - register machines: operations are performed on a fixed finite set of global locations ("registers") (e.g.: SPIM);
  - stack machines: operations are performed on the top of a stack of operands (e.g.: JVM).

---

### Byte Code

- Operations encoded as small integers (~1 byte).
- The interpreter uses the opcode to index into a table of code addresses:

```c
while ( TRUE ) {
    byte op = *ip;
    switch ( op ) {
        case ADD: ... perform addition; break;
        case SUB: ... perform subtraction; break;
        ... 
    }
}
```

- Advantages: simple, portable.
- Disadvantages: inefficient.
Direct Threaded Code

- Indexing through a jump table (as with byte code) is expensive.
- **Idea**: Use the address of the code for an operation as the opcode for that operation.
  - The opcode may no longer fit in a byte.

```c
while ( TRUE ) {
    addr op = *ip;
    goto *op; /* jump to code for the operation */
}
```

- More efficient, but the interpreted code is less portable.

Byte Code vs. Threaded Code

**Control flow behavior:**

**byte code:**

- Program: Byte code program
- Interpreter: Interprets byte code
- Data: Exposed

**threaded code:**

- Program: Threaded code program
- Interpreter: Interprets byte code
- Data: Exposed
- Code for interpreter operations linked
Indirect Threaded Code

- Intermediate in portability, efficiency between byte code and direct threaded code.
- Each opcode is the address of a word that contains the address of the code for the operation.
  - Avoids some of the costs associated with translating a jump table index into a code address.

```c
while ( TRUE ) {
    addr op = *ip;
    goto **op; /* jump to code for the operation */
}
```


Example

<table>
<thead>
<tr>
<th>program operations</th>
<th>memory address</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>10000</td>
<td>add:...</td>
</tr>
<tr>
<td>mul</td>
<td>11000</td>
<td>sub:...</td>
</tr>
<tr>
<td>sub</td>
<td>12000</td>
<td>mul:...</td>
</tr>
<tr>
<td>mul</td>
<td>13000</td>
<td>div:...</td>
</tr>
<tr>
<td>add</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>byte code</th>
<th>Op Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>index</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>17</td>
<td>27</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>byte code</th>
<th>Op Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>12000</td>
<td>11000</td>
</tr>
<tr>
<td>12000</td>
<td>10000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>direct threaded</th>
<th>indirect threaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>contents</td>
</tr>
<tr>
<td>6000</td>
<td>6000</td>
</tr>
<tr>
<td>6008</td>
<td>10000</td>
</tr>
<tr>
<td>6004</td>
<td>6004</td>
</tr>
<tr>
<td>6008</td>
<td>11000</td>
</tr>
<tr>
<td>6000</td>
<td>6000</td>
</tr>
<tr>
<td>6012</td>
<td>13000</td>
</tr>
</tbody>
</table>
Handling Operands 1: Stack Machines

- Used by Pascal interpreters ('70s and '80s); resurrected in the Java Virtual Machine.

- **Basic idea:**
  - operands and results are maintained on a stack;
  - operations pop operands off this stack, push the result back on the stack.

- **Advantages:**
  - simplicity
  - compact code.

- **Disadvantages:**
  - code optimization (e.g., utilizing hardware registers effectively) not easy.

Stack Machine Code

- The code for an operation ‘\( op \) \( x_1, \ldots, x_n \)’ is:
  
  ```java
  push x_n
  ...
  push x_1
  op
  ```

- **Example:** JVM code for ‘\( x = 2*y - 1 \)’:
  
  ```java
  icanst 1 /* push the integer constant 1 */
  iload y /* push the value of the integer variable y */
  icanst 2
  imul /* after this, stack contains: (2*y) */
  isub /* pop stack top, store to integer variable x */
  istore x /* store x */
  ```
Generating Stack Machine Code

Essentially just a post-order traversal of the syntax tree:

```c
void gencode(struct syntaxTreeNode *tnode)
{
    if (IsLeaf(tnode)) { ... }
    else {
        n = tnode->n_operands;
        for (i = n; i > 0; i--) {
            gencode(tnode->operand[i]); /* traverse children first */
        } /* for */
        gen_instr(opcode_table[tnode->op]); /* then generate code for the node */
    } /* if [else] */
}
```

Handling Operands 2: Register Machines

- **Basic idea:**
  - Have a fixed set of “virtual machine registers;”
  - Some of these get mapped to hardware registers.

- **Advantages:**
  - Potentially better optimization opportunities.

- **Disadvantages:**
  - Code is less compact;
  - Interpreter becomes more complex (e.g., to decode VM register names).
Just-in-Time Compilers (JITs)

- **Basic idea**: compile byte code to native code during execution.
- **Advantages**:
  - original (interpreted) program remains portable, compact;
  - the executing program runs faster.
- **Disadvantages**:
  - more complex runtime system;
  - performance may degrade if runtime compilation cost exceeds savings.

Improving JIT Effectiveness

- **Reducing Costs**:
  - incur compilation cost only when justifiable;
  - invoke JIT compiler on a per-method basis, at the point when a method is invoked.

- **Improving benefits**:
  - some systems monitor the executing code;
  - methods that are executed repeatedly get optimized further.
Method Dispatch: vtables

- vtables ("virtual tables") are a common implementation mechanism for virtual methods in OO languages.
- The implementation of each class contains a vtable for its methods:
  - each virtual method \( f \) in the class has an entry in the vtable that gives \( f \)'s address.
  - each instance of an object gets a pointer to the corresponding vtable.
  - to invoke a (virtual) method, get its address from the vtable.

VMs with JITs

- Each method has vtable entries for:
  - byte code address;
  - native code address.
- Initially, the native code address field points to the JIT compiler.
- When a method is first invoked, this automatically calls the JIT compiler.
- The JIT compiler:
  - generates native code from the byte code for the method;
  - patches the native code address of the method to point to the newly generated native code (so subsequent calls go directly to native code);
  - jumps to the native code.
For a JIT to improve performance, the benefit of compiling to native code must offset the cost of doing so.

*E.g., JIT compiling infrequently called straight-line code methods can lead to a slowdown!*

- We want to JIT-compile only those methods that contain frequently executed (“hot”) code:
  - methods that are called a large number of times; or
  - methods containing loops with large iteration counts.

### Identifying frequently called methods:
- count the number of times each method is invoked;
- if the count exceeds a threshold, invoke JIT compiler.
- (In practice, start the count at the threshold value and count down to 0: this is slightly more efficient.)

### Identifying hot loops:
- modify the interpreter to “snoop” the loop iteration count when it finds a loop, using simple bytecode pattern matching.
- use the iteration count to adjust the amount by which the invocation count for the method is decremented on each call.
**Typical JIT Optimizations**

Choose optimizations that are cheap to perform and likely to improve performance, e.g.:

- inlining frequently called methods:
  
  *consider both code size and call frequency in inlining decision.*

- exception check elimination:
  
  *eliminate redundant null pointer checks, array bounds checks.*

- common subexpression elimination:
  
  *avoid address recomputation, reduce the overhead of array and instance variable access.*

- simple, fast register allocation.