LECTURE 2

Compilers and Interpreters

COMPILATION AND INTERPRETATION

Programs written in high-level languages can be run in two ways.

- **Compiled** into an executable program written in machine language for the target machine.
- Directly **interpreted** and the execution is simulated by the interpreter.

COMPILATION AND INTERPRETATION

Let’s say we have the following statement:

```
A[i][j] = 1;
```

How can we execute this statement?

COMPILATION AND INTERPRETATION

```
A[i][j] = 1;
```

An Approach:

- Create a software environment that understands 2-dimensional arrays and the language.
- To execute the statement, it just puts 1 in the array entry `A[i][j]`.
- This is **interpretation** since the software environment understands the language and performs the operations specified by interpreting the statements.

COMPILATION AND INTERPRETATION

```
A[i][j] = 1;
```

Another Approach:

- Translate the statements into native machine language (or assembly language) and then run the program.
- This is **compilation**, g++ produces the following assembly for this statement:

```
salq    $2, %rax
addq    %rcx, %rax
leaq    0(,%rax,4), %rdx
addq    %rdx, %rax
salq    $2, %rax
addq    %rsi, %rax
movl    $1, A(,%rax,4)
```

COMPILATION AND INTERPRETATION

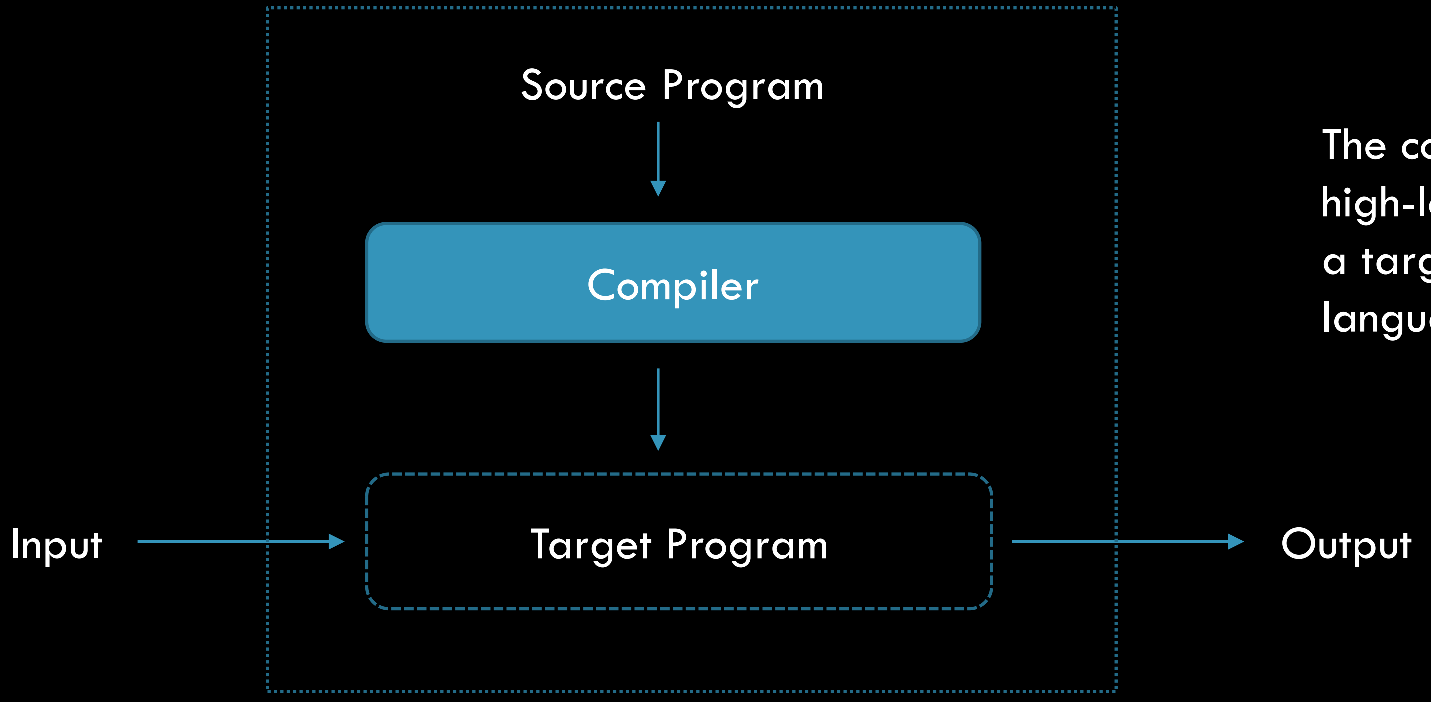
- How is a C++ program executed on linprog?
  - `g++ try.cpp` → compiling the program into machine code.
  - `./a.out` → running the machine code.
- How is a python program executed?
  - `python try.py`
  - The program just runs, no compilation phase.
  - The program `python` is the software environment that understands python language.
  - The program `try.py` is executed (interpreted) within the environment.
- In general, which approach is more efficient?

COMPILATION AND INTERPRETATION

- How is a C++ program executed on linprog?
  - `g++ try.cpp` → compiling the program into machine code.
  - `./a.out` → running the machine code.
- How is a python program executed?
  - `python try.py`
  - The program just runs, no compilation phase.
  - The program `python` is the software environment that understands python language.
  - The program `try.py` is executed (interpreted) within the environment.
- In general, which approach is more efficient?
  - Compilation is *always* more efficient!
  - Interpretation provides more functionality.

COMPILATION

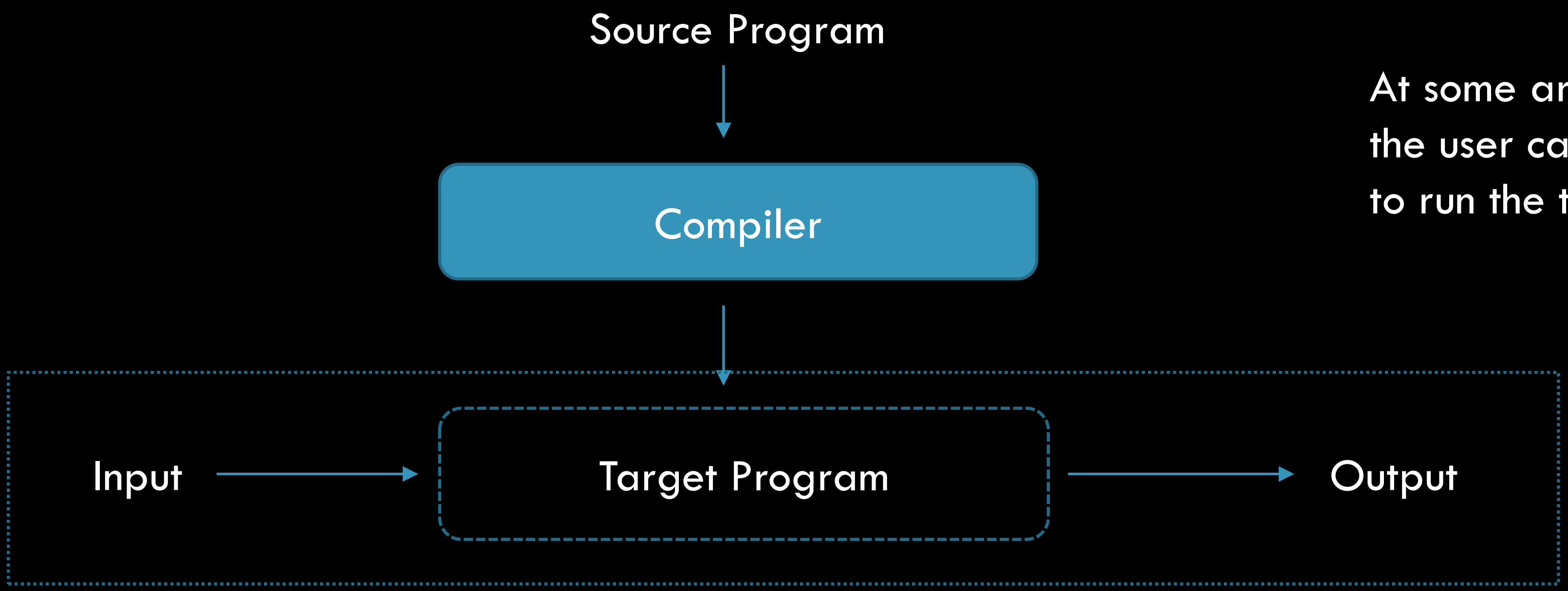
At the highest level of abstraction, compiling looks like this:



The compiler translated the high-level source program into a target program in the machine's language (object code).

COMPILERS

At the highest level of abstraction, compiling looks like this:



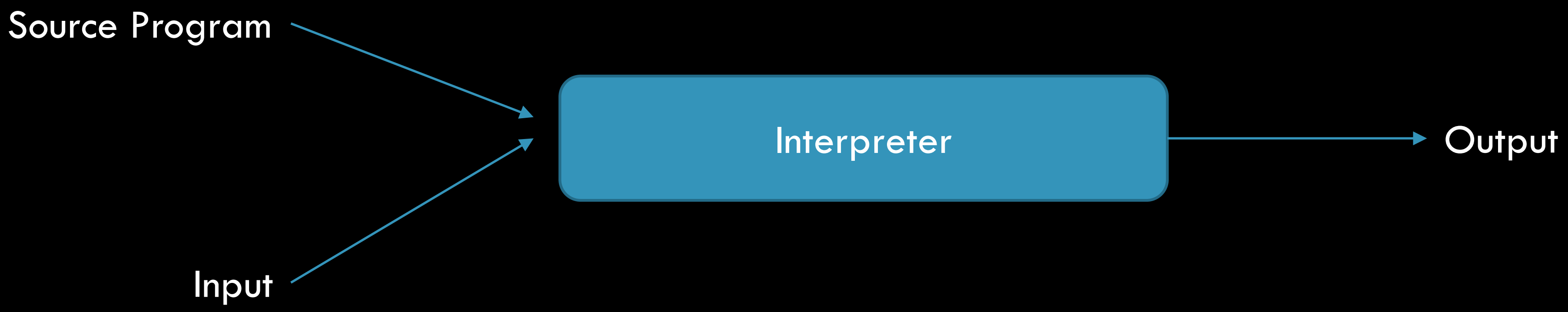
At some arbitrary later time,
the user can tell the operating system
to run the target program.

WHAT IS A COMPILER?

The compiler itself is also a machine language program, typically created by compiling some other high-level program.

INTERPRETERS

Interpretation generally looks like this:



Interpreters are necessary for the execution of the application. Interpreters essentially create a VM whose machine language *is* the high-level programming language.

COMPILATION VS. INTERPRETATION

Compilers attempt to make decisions at compile time to avoid them at run time.

- Type checking at compile time vs. runtime.
- Static allocation.
- Static linking.
- Code optimization.

Compilation leads to better performance in general.

- Allocation of variables without variable lookup at run time.
- Aggressive code optimization to exploit hardware features.

COMPILATION VS. INTERPRETATION

So why use interpreted languages?

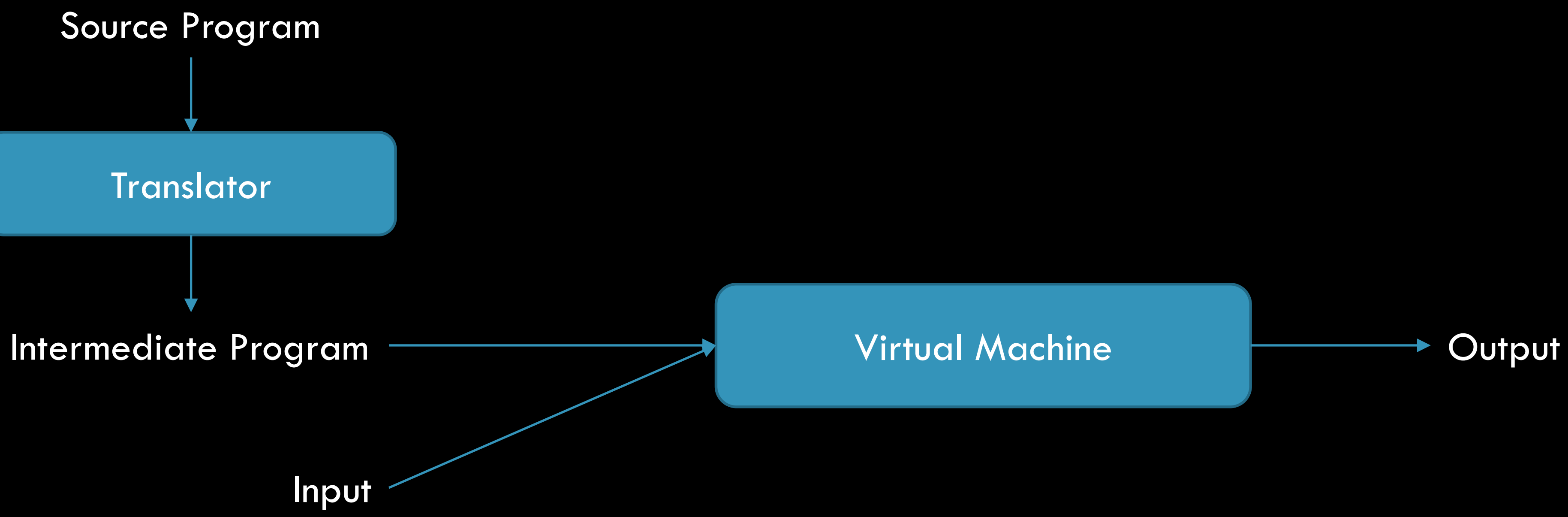
Interpretation leads to greater flexibility, easier debugging, and “better” features.

- Fundamental characteristics can be decided at run time.
  - Example: `some_input = raw_input(“Please type something: ”)` ← perfectly valid Python
- Lisp and Prolog can write new pieces of code and execute them on the fly.

MIXING COMPILATION AND INTERPRETATION

How do you choose?
Don't worry – you don't have to. Kinda.

Typically, most languages are implemented using a mixture of both approaches.



VIRTUAL MACHINES

Virtual machines are typically software emulations of a machine.

- System virtual machines emulate entire platforms.
- Language virtual machines support a single process. ← We're mostly concerned with these.

An important example is the Java Virtual Machine (JVM).

- Can execute any executable that is compiled into Java bytecode.

Technically, your CPU can be viewed as an implementation in hardware of a virtual machine (e.g. bytecode can be executed in hardware).

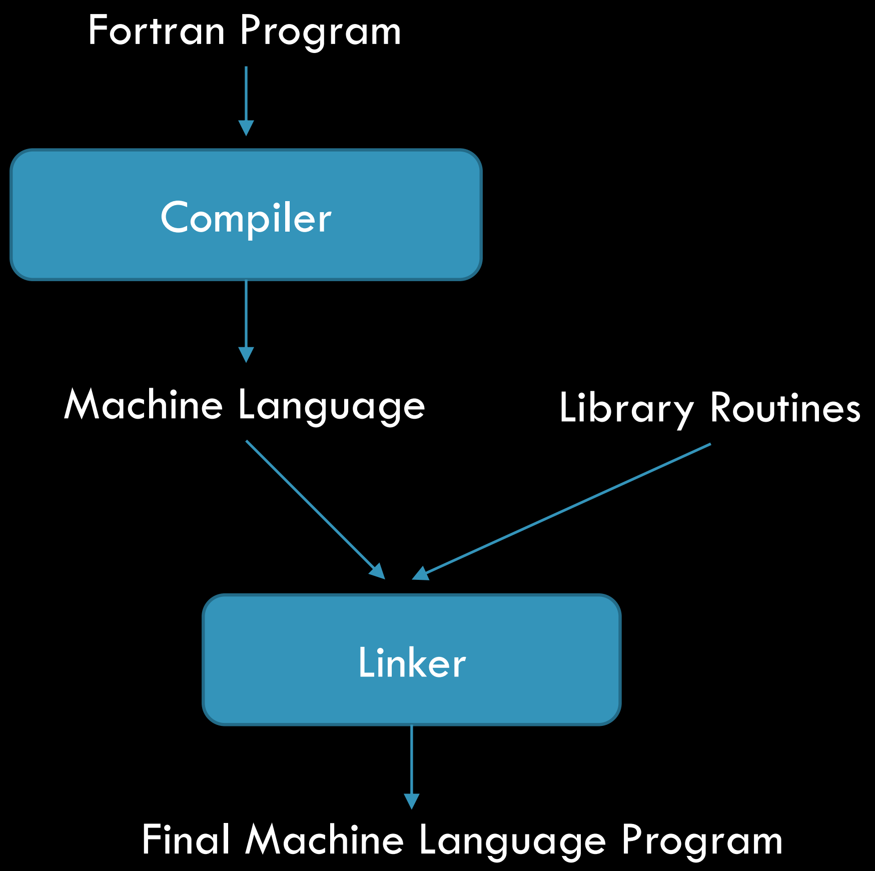
MIXING COMPILATION AND INTERPRETATION

Practically speaking, there are two aspects that distinguish what we consider "compilation" from "interpretation".

- Thorough Analysis
  - Compilation requires a thorough analysis of the code.
- Non-trivial Transformation
  - Compilation generates intermediate representations that typically do not resemble the source code.

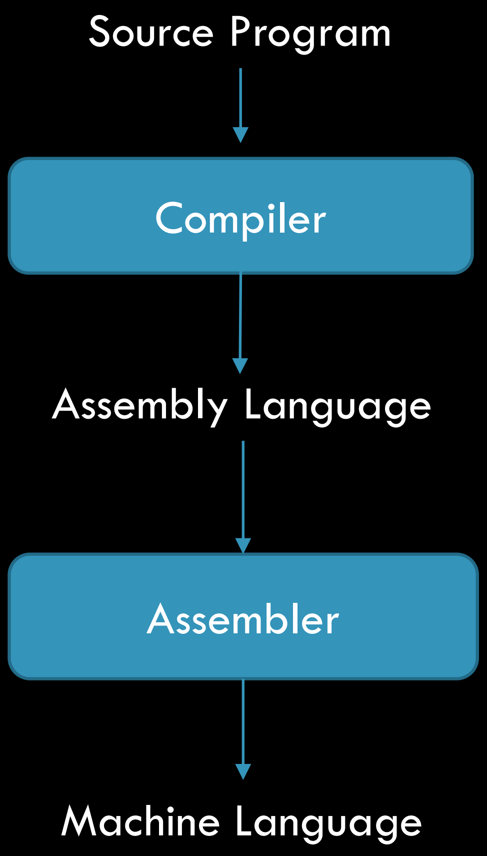
PRACTICAL IMPLEMENTATION STRATEGIES

- Preceprocessing
  - Initial translation step.
  - Slightly modifies source code to be interpreted more efficiently.
  - Removing comments and whitespace, grouping characters into tokens, etc.
  - C preprocessor can modify portions of code itself → *conditional compilation.*
- Linking
  - Linkers merge necessary library routines to create the final executable.
  - Fortran implementations come closest to pure compilation with the exception of a linking step.



PRACTICAL IMPLEMENTATION STRATEGIES

- Post-Compilation Assembly
  - Many compilers translate the source code into assembly rather than machine language.
  - Changes in machine language won't affect source code.
  - Assembly is easier to read (for debugging purposes).
- Source-to-source Translation
  - Compiling source code into another high-level language.
  - Early C++ programs were compiled into C, which was compiled into assembly.



PRACTICAL IMPLEMENTATION STRATEGIES

- Bootstrapping
  - What comes first, the language or the compiler?
  - Let’s say you want to build a compiler for Java that is written in Java (*self-hosting*), but we only have a C compiler.
  - Write a very simple compiler for a small subset of Java in a small subset of C.
  - Hand-translate the compiler into Java.
  - Run the translated code through the C-written compiler.
  - Now you have a Java compiler written in Java.
  - Repeat, extending the compiler to accept a larger subset of Java.

PRACTICAL IMPLEMENTATION STRATEGIES

- Dynamic and Just-in-time Compilation
  - Dynamic compilation is the delay of compilation until the last possible moment.
  - JIT is a subset of Dynamic Compilation and combines traditional compilation with interpretation (only the source code → bytecode occurs ahead of time).
  - JIT compilation combines the speed of compiled code with the flexibility of interpretation, with the overhead of both methods combined.

COMPILATION

So, clearly compilation is not so rigidly defined as we might have expected.

As stated before, it suffices to say that compilation is the translation of a nontrivial language to another non-trivial language, with thorough analysis of the input.

INTEGRATED DEVELOPMENT ENVIRONMENTS

- With all that said, programming tools function together in concert.
  - Editors
  - Compilers/Preprocessors/Interpreters
  - Debuggers
  - Emulators
  - Assemblers
  - Linkers
- Advantages
  - Tools and compilation stages are hidden.
    - You’ve been programming for a while now, did you know about all these compilation methods?
  - Automatic source-code dependency checking.
  - Debugging made simpler.
  - Editor with search facilities.

NEXT LECTURE

Compiler Phases