There are a few main approaches to running code:

The Compiler Model

think: C, C++, Fortran, etc.
gcc's process takes c source code, and runs like so:

1. Preprocessing
   (a) lexing → tokens
   (b) parsing → parse tree
   (c) checking (identifiers, type) → checked parse tree
   (d) Machine Independent Code generation → MIC
   (e) optimization → optimized intermediate MIC
2. Compilation
   (a) machine code generation → machine dependent code
   (b) optimization → assembly language
3. Assembly
   (a) assembly → machine code (.o)
4. Linking
   (a) linking (ld) → a.out
5. Running
   (a) loading by kernel → run

Each of these steps can be broken into a separate program using gcc options and tools.
Compilers can make programs quite difficult to debug, since code is translated away from the source code.
For instance, consider the following two translations:
i *= 2 → shl %rax
1 = a/b; r = a % b; → ldivq %rdl
By the above, it can be seen that some translations are more direct than others.
The transformations compilers can perform include:

- aggregation of multiple lines of source code into one machine instructions
- splitting of a line of source code into multiple machine instructions
- swapping the order of machine commands

Since debugging machine code can be incredibly difficult, we developed:

The Interpreter Model

The program is left in a form near to the source code and interpreted/run directly.
The form of the code is designed for an abstract machine and human-readable.
Code is checked prior to translation.
The interpreter is written in a high-level language and thus easy to implement.

As we can see, interpreters have many advantages over compilers, so why do we use compilers?
When running code, an interpreter must walk through a tree or load bytes from MIC.
This can result in up to 10 machine instructions for each line of source code.
In practice, much of this is optimized away, but the principle stands; we wish to improve this.

Integrated Development Environment

think: Eclipse, Emacs, etc.
The original was Smalltalk by Xerox, which:

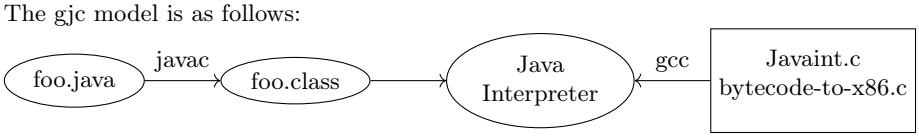
- handles dependencies and partial recompilation
- keeps "one big happy program", including:
  - memory management
  - exception handling

- dumping core
- parallelism

- interacts with and controls the screen, keyboard, and mouse
- reads and works either directly from source code, or from a form very close to it, which allows
  - direct debugging of source code
  - continuous execution of program while editing

Java Virtual Machine (JVM)

The gjc model is as follows:



The steps

- the interpreter uses bytecode-to-x86.c to profile the code
  ≡ the compilation of recorded hotspots to machine code
- jump to machine code rather than fetching interpreted instructions
- progressively compile learned hot spots that pop up during execution

In a sense, the profiler acts like the backend of a compiler. This has a few advantages:

- space – shared code need only be stored once rather than once per program
- time – recompilation is not necessary on change of source code
- space – unused parts of the library are not unnecessarily loaded

This is called a **just-in-time/hotspot compiler**

Conclusion

Real life is unsurprisingly more complicated; this behavior exists on a spectrum:

- pure interpreter – the intermediate language IS the source code
- tokenization – whitespace and comments are removed (the most common level)
- partial compilation — intermediate code could either be compiled or run directly on an interpreter
- full compilation – code is fully compiled; the intermediate code is machine code

Emacs and eclipse both fall somewhere in the middle of this spectrum.
The JVM approach is used for Javascript on browsers, which requires varying code by machine.
This is no big deal – we already needed different machine code anyway.

IDE's and Virtual Machines require one more functionality: **Dynamic Linking**
We take a piece of a program/file and link it into a running program. Examples include:

- emacs takes shared machine code object files and links it using emacs Lisp
- JVM and Javascript do the same thing with their own native methods

In effect, this creates self modifying code, which is an idea fundamental to CS! See: AI, the stack, etc.)
There are two ways to dynamically link:

1. complete linkage on load time This is safe, but much slower.
2. load functions on usage This can be risky, since
   (a) functions may change between calls
   (b) malicious or bad code may cause errors

Ideally, we would like the power of self modifying code without the risk.
This class's focus so far has been on imperative languages, which are command execution based.
Our attempt to improve dynamic linking is by using functional languages.
There are based in evaluation of expressions.
These avoid the side effects and cache invalidation of imperative languages.
This gives us the advantage of building a program on the fly, without the dangers of dynamic linking!