From Source Code to Executable Code

program gcd(input, output);
var i, j: integer;
begin
  read(i, j);
  while i <> j do
    if i > j then i := i - j;
    else j := j - i;
  writeln(i)
end.
**Phases of Compilation**

- Character stream → Scanner (lexical analysis)
- Token stream → Parser (syntax analysis)
- Parse tree → Semantic analysis and intermediate code generation
- Abstract syntax tree or other intermediate form → Machine-independent code improvement (optional)
- Modified intermediate form → Target code generation
- Assembly or machine language, or other target language → Machine-specific code improvement (optional)
- Modified target language → Symbol table

**Compiler Structure**

- Lexical, syntax and semantic analyses are the **front-end** of a compiler
  - These **phases** serve to figure out the meaning of the program
- The rest of the phases are considered part of the **back-end** of a compiler
  - They are responsible for the generation of the target program
Phases of Compilation

- The first three phases are language-dependent
- The last two are machine-dependent
- The middle two dependent on neither the language nor the machine

Example

```pascal
program gcd(input, output);
var i, j: integer;
begin
  read(i, j);
  while i <> j do
    if i > j then i := i - j;
    else j := j - i;
  writeln(i)
end.
```
Example
Syntax Tree and Symbol Table

```
program (5)
    read (6)
    read (3)
    write (4)
    while (3) (7)
        write (4)
        write (9)
    null
```

<table>
<thead>
<tr>
<th>Index</th>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTEGER</td>
<td>type</td>
</tr>
<tr>
<td>2</td>
<td>INPUT</td>
<td>type</td>
</tr>
<tr>
<td>3</td>
<td>INTEGER</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>OUTPUT</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>prog</td>
<td>program</td>
</tr>
<tr>
<td>6</td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>j</td>
<td>1</td>
</tr>
</tbody>
</table>

Phases of Compilation

- **Intermediate code generation** transforms the abstract syntax tree into a less hierarchical representation: a control flow graph.
**Example**

**Control Flow Graph**

- **Basic blocks** are maximal-length set of sequential operations
  - Operations on a set of virtual registers
    » Unlimited
    » A new one for each computed value
- **Arcs represent** interblock control flow

**Phases of Compilation**

- **Machine-independent code improvement** performs a number of transformations:
  - Eliminate redundant loads stores and arithmetic computations
  - Eliminate redundancies across blocks
Phases of Compilation

- **Target Code Generation** translates block into the instruction set of the target machine, including branches for the arc
- It still relies in the set of virtual registers

- **Machine-specific code improvement** consists of:
  - Register allocation (mapping of virtual register to physical registers and multiplexing)
  - Instruction scheduling (fill the pipeline)
Compilation Passes

- A pass of compilation is a phase or sequence of phases that is serialized with respect to the rest of the compilation.
  - It may be written as separate program that relies on files for input and output.
- Two-pass compilers are very common.
  - Front-end and back-end passes, or intermediate code generation and global code improvement.
- Most compilers generate assembly, so the assembler behaves as an extra pass.
- Assembly requires linking that may take place at compilation, load or run-time.

Why are compilers divided in passes?

- Sharing the front-end among the compilers of more than one machine.
- Sharing the back-end among the compilers of more than one language.
- Historically, passes help reducing memory usage.
Intermediate Forms

- Front-end and back-end are linked using an abstract representation known as the Intermediate Format (IF)
  - The IF is propagated through the back-end phases
- They classified according to their level of machine dependence
- High-level IFs are usually trees or directed acyclic graphs that capture the hierarchy of the program
  - They are useful for machine-independent code improvement, interpretation and other operations

Intermediate Forms

Stack-based Language

- Stack-based language are another type of IFs
  - E.g. JVM, Pascal’s P-code
- They are simple and compact
  - They resemble post-order tree enumeration
- Operations
  - Take their operands from an implicit stack
  - Return their result to an implicit stack
- These languages tend to make language easy to port and the result code is very compact
  - Ideal for network transfer of applets
Java Virtual Machine

- JVM spec

- Tutorial

Reading Assignment

- Read Scott
  - Sect. 9 Intro
  - Sect. 9.1
  - Sect. 9.2 intro, and glance at IDL and RTL