Goals

- In this lecture, we will discuss how to translate source code into machine executable code.
Compilation and Interpretation

- There are two primary methods for translating high-level code:
  - Compilation
  - Interpretation
Compilation

Source Program

Compiler

Input → Target Program → Output
The “target program” is called the object code.
Compilation

Source Program

You translate \textit{once} and \textit{run many times}.

Input  \rightarrow  Target Program  \rightarrow  Output
Interpretation

Source Program

Input

Interpreter

Output
Interpretation

You translate for each run.
Comparison

The University of North Carolina at Chapel Hill

Source

Compiler

Target

Out

Source

Interpreter

Out

In
Compilers are usually faster than interpreters.
Interpreters are usually **more flexible** and **easier to debug** than compilers.
Compilation and Interpretation

The Virtual Machine acts as an interpreter.
The translator can be a compiler or an interpreter. It is considered to be a compiler if:

1. There is a thorough analysis of the program
2. The transformation is non-trivial.
Compilation and Interpretation

Source Program

Translator

Intermediary Program

Virtual Machine

Input

This is exactly the process that Java uses.

Output
The University of North Carolina at Chapel Hill

Linking

Source Program

Compiler

Incomplete machine language

Library Routines

Linker

Machine language program
The Linker is used to connect subroutines that are not contained in the original code.
Linking

Source Program

Compiler

This is used by Fortran

Incomplete machine language

Library Routines

Linker

Machine language program

This is used by Fortran
Preprocessing

Source Program

Preprocessor

Modified source program
Preprocessing

For example, this may be used to remove comments or use replacement macros.

Preprocessor

Modified source program
On the note about macros, consider the macro
#define FALSE 0.
This code will replace all instances of the word “FALSE” with the value 0.
Tombstone Diagrams

- Useful for graphically representing programs and translators.
  - Program (in C):
    - sort
    - C
  - Machine (x86):
    - x86
  - Interpreter (for Perl, impl. in x86)
    - Perl
    - x86
  - Translator (C to x86, impl. in x86)
    - C
    - x86
    - x86
Tombstone Diagrams

• Example: Compiling a C program to run on x86
Pascal came shipped with three things:

- A Pascal to P-Code Compiler, written in Pascal.
- A P-code interpreter, in Pascal.
- A Pascal to P-Code Compiler, written in P-Code.
Pascal came shipped with three things:

- A Pascal to P-Code Compiler, written in Pascal.
- A P-code interpreter, in Pascal.
- A Pascal to P-Code Compiler, written in P-Code.

P-Code is a very simple language that can easily be translated into any machine language.
Bootstrapping

• Pascal came shipped with three things:
  • A Pascal to P-Code Compiler, written in Pascal.
  • A P-code interpreter, in Pascal.
  • A Pascal to P-Code Compiler, written in P-Code.
Bootstrapping

- Pascal came shipped with three things:
  - A Pascal to P-Code Compiler, written in Pascal.
  - The interpreter is **hand translated** into a supported language.
  - A Pascal to P-Code Compiler, written in P-Code.
Bootstrapping

• Pascal came shipped with three things:
  • A Pascal to P-Code Compiler, written in Pascal.

The interpreter is **hand translated** into a supported language. And then compiled into machine code.
Bootstrapping

• To get a simple (but slow) compiler, one could use the “Pascal to P-Code compiler, in P-Code” and the “P-Code to machine lang. interpreter” to compile Pascal code.
To get a simple (but slow) compiler, one could use the “Pascal to P-Code compiler, in P-Code” and the “P-Code to machine lang. interpreter” to compile Pascal code.

This interpreter is used to run the **Pascal to P-Code** compiler and the program.
Bootstrapping (Faster)

- To create a faster compiler, we modify the “Pascal to P-Code compiler, in \textbf{Pascal}” so that it is a “Pascal to machine language compiler, in \textbf{Pascal}.”
Bootstrapping (Faster)

This is **MUCH HARDER** than creating a P-Code to Machine language interpreter.

Diagram: Pasc -> Pcode -> Manual Transformation -> x86
Now construct the “Pascal to machine lang. compiler, in P-Code.”
Then run the “Pascal to machine lang. compiler, in **Pascal**” through the “**P-Code**” version to produce the “**Machine lang.**” version.
Bootstrapping (Faster)

End product: a Pascal compiler!
Compiling

Character Stream → Scanner (lexical analysis)

Token Stream → Parser (syntax analysis)

Parse Tree → Semantic analysis & intermediate code gen.

Abstract syntax tree → Machine-independent optimization (optional)

Modified intermediate form → Target code generation.

Machine language → Machine-specific optimization (optional)

Modified target language → Symbol Table
Example Program GCD

program gcd(input, output);
var i, j: integer;
begin
  read(i,j); // get i & j from read
  while i<>j do
    if i>j then i := i-j
    else j := j-1;
  writeln(i)
end.
Lexical Analysis

- Recognize structures without regard to meaning and groups them into **tokens**.

- The purpose of the scanner is to simplify the parser by reducing the size of the input.
Syntax Analysis

• Parsing organizes the tokens into a context-free grammar (i.e., syntax).

```
program gcd ( input , output ) ;
```

```
program
descendants

id(GCD) ( id(INPUT) more_ids ) ; block

, id(OUTPUT) more_ids
empty

Rest of code
```
Syntax Analysis

• Parsing organizes the tokens into **a context-free grammar** (i.e., syntax).

```plaintext
program gcd ( input , output ) ;
```

The Syntax analysis catches **all malformed statements**

```plaintext
id(GCD) ( id(INPUT) more_ids ) ;
```

```plaintext
, id(OUTPUT) more_ids
```

```plaintext
empty
```

```plaintext
block
```

```plaintext
Rest of code
```
Syntax Analysis

• Parsing organizes the tokens into a **context-free grammar**.

The parse tree is sometimes called a **concrete syntax tree** because it contains how all tokens are derived...

```plaintext
id(GCD) ( id(INPUT) more_ids ) ; block
 , id(OUTPUT) more_ids
 empty

Rest of code
```
Syntax Analysis

- Parsing organizes the tokens into a context-free grammar (i.e., syntax).

However, much of this information is extraneous for the “meaning” of the code (e.g., the only purpose of “;” is to end a statement).
Semantic Analysis

• Semantic analysis discovers the **meaning of a program** by creating an **abstract syntax tree** that removes “extraneous” tokens.

• To do this, the analyzer builds & maintains a **symbol table** to map identifiers to information known about it. (i.e., scope, internal structure, etc...)

• By using the symbol table, the semantic analyzer can **catch problems not caught by the parser**. For example,
  
  • Identifiers are declared before used
  
  • subroutine calls provide correct number and type of arguments.
Semantic Analysis

- Not all semantic rules can be checked at compile time.
  - Those that can are called **static** semantics of the language.
  - Those that cannot are called **dynamic** semantics of the language. For example,
    - Arithmetic operations **do not overflow**.
    - Array subscripts expressions **lie within the bounds of the array**.
Example Program GCD

```pascal
program gcd(input, output);
var i, j: integer;
begin
  read(i,j); // get i & j from read
  while i<>j do
    if i>j then i := i-j
    else j := j-1;
  writeln(i)
end.
```
### Semantic Analysis

```
program
  id(GCD) ( id(INPUT) more_ids ) ;
```

<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>TEXTFILE</td>
<td>type</td>
</tr>
<tr>
<td>3</td>
<td>INPUT</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>OUTPUT</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>GCD</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>
Target code generation

- Code generation takes the abstract syntax tree and the symbol table to produce machine readable code.
- *Simple* code follows directly from the abstract syntax tree and symbol table.
Optimization

• The process so far will produce correct code, but it may not be fast.

• Optimization will adjust the code to improve performance.
  
  • A possible machine-indp. optimization would be to keep the variables $i$ and $j$ in registers throughout the main loop.
  
  • A possible machine-spec. optimization would be to assign the variables $i$ and $j$ to specific registers.