• Compilation and interpretation schemes
• Cross compilation
• Bootstraping
• Compilers
Implementing a Programming Language

- **L** high level programming language
- **M_L** abstract machine for L
- **M_o** host machine

**Pure Interpretation**
- **M_L** is interpreted over **M_o**
- Not very efficient, mainly because of the interpreter (fetch-decode phases)

**Pure Compilation**
- Programs written in L are translated into equivalent programs written in L_0, the machine language of **M_o**
- The translated programs can be executed directly on **M_o**
  - **M_L** is not realized at all
  - Execution more efficient, but the produced code is larger

- Two limit cases that almost never exist in reality
Pure Interpretation

- Program $P$ in $L$ as a partial function on $D$:
  \[ P^L : D \rightarrow D \]
- Set of programs in $L$: $\text{Prog}^L$

\[ I^L_0 : (\text{Prog}^L \times D) \rightarrow D \text{ such that } I^L_0(P^L, \text{Input}) = P^L(\text{Input}) \]

The interpreter defines a function
Pure [cross] Compilation

A compiler from $L$ to $LO$ defines a function

$$\mathcal{CL},\mathcal{LO} : \text{Prog}^L \rightarrow \text{Prog}^{LO}$$

such that if

$$\mathcal{CL},\mathcal{LO}(P^L) = P^{CLO},$$

then for every input we have

$$P^L(\text{Input}) = P^{CLO}(\text{Input})$$

---

Diagram:

- Program written in $L$
- Compiler from $L$ to $LO$
- Program written in $LO$
- Output data

Flow:

- Execution on $MA$
- Execution $MO$

Machine:

- Abstract machine $MA$
- Host machine $MO$
Compilers versus Interpreters

• Compilers efficiently fix decisions that can be taken at compile time to avoid to generate code that makes this decision 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

• Interpretation facilitates interactive debugging and testing
  – Interpretation leads to better diagnostics of a programming problem
  – Procedures can be invoked from command line by a user
  – Variable values can be inspected and modified by a user
Compilation + Interpretation

• All implementations of programming languages use both. At least:
  – Compilation (= translation) from external to internal representation
  – Interpretation for I/O operations (runtime support)

• Can be modeled by identifying an Intermediate Abstract Machine $M_I$ with language $L_I$
  – A program in $L$ is compiled to a program in $L_I$
  – The program in $L_I$ is executed by an interpreter for $M_I$
The real situation for the implementation of a high-level language is therefore that shown in Fig. 1.6. Let us assume, as above, that we have a language $L$ that has to be implemented and assume also that a host machine $M_0$ exists which has already been constructed. Between the machine $M$ that we want to implement and $M_0$ we introduce an intermediate machine $M_i$.

- The “pure” schemes as limit cases
- Let us sketch some typical implementation schemes...
Virtual Machines as Intermediate Abstract Machines

• Several language implementations adopt a compilation + interpretation schema, where the Intermediate Abstract Machine is called Virtual Machine

• Adopted by Pascal, Java, Smalltalk-80, C#, functional and logic languages, and some scripting languages
  – Pascal compilers generate P-code that can be interpreted or compiled into object code
  – Java compilers generate bytecode that is interpreted by the Java virtual machine (JVM)
  – The JVM may translate bytecode into machine code by just-in-time (JIT) compilation
Compilation and Execution on Virtual Machines

- Compiler generates intermediate program
- Virtual machine interprets the intermediate program

Source Program → Compiler → Intermediate Program

Compile on X → Run on VM

Input → Virtual Machine → Output

Run on X, Y, Z, …

• Portability!
Pure Compilation and Static Linking

- Adopted by the typical Fortran systems
- Library routines are separately linked (merged) with the object code of the program

```
extern printf();
```

```
_printf
_fget
_fscan
...
```

```
Binary
Executable
```
Compilation, Assembly, and Static Linking

- Facilitates debugging of the compiler
Compilation, Assembly, and Dynamic Linking

• Dynamic libraries (DLL, .so, .dylib) are linked at run-time by the OS (via stubs in the executable)
Preprocessing

- Most C and C++ compilers use a preprocessor to import header files and expand macros.

```
#include <stdio.h>
#define N 99
...
for (i=0; i<N; i++)
```

```
for (i=0; i<99; i++)
```

```
Assembly or Object Code
```
The CPP Preprocessor

• Early C++ compilers used the CPP preprocessor to generated C code for compilation
Compilers, graphically

• Three languages involved in writing a compiler
  – Source Language (S)
  – Target Language (T)
  – Implementation Language (I)

• T-Diagram:

  • If I = T we have a Host Compiler
  • If S, T, and I are all different, we have a Cross-Compiler
Composing compilers

• Compiling a compiler we get a new one: the result is described by composing T-diagrams

\[ \text{Example: } S \quad \text{Pascal} \\
I \quad I \quad C \\
M \quad M \quad 68000 \]

• A compiler of S to M can be written in any language having a host compiler for M
Composing compilers

• Compiling a compiler we get a new one: the result is described by composing T-diagrams

• The shape of the basic transformation, in the most general case, is the following:

    ![Diagram]

• Note: by writing this transformation, we implicitly assume that we can execute programs written in $\mathcal{M}$
Bootstrapping

• **Bootstrapping**: techniques which use partial/inefficient compiler versions to generate complete/better ones
• Often compiling a translator programmed in its own language
• Why writing a compiler in its own language?
  – it is a non-trivial test of the language being compiled
  – compiler development can be done in the higher level language being compiled.
  – improvements to the compiler’s back-end improve not only general purpose programs but also the compiler itself
  – it is a comprehensive consistency check as it should be able to reproduce its own object code
Compilers: Portability Criteria

• Portability
  – Retargetability
  – Rehostability

• A **retargetable** compiler is one that can be modified easily to generate code for a new target language

• A **rehostable** compiler is one that can be moved easily to run on a new machine

• A portable compiler may not be as efficient as a compiler designed for a specific machine, because we cannot make any specific assumption about the target machine
Using Bootstrapping to port a compiler

- We have a host compiler/interpreter of L for M
- Write a compiler of L to N in language L itself

Example:
L Pascal
M P-code

Diagram showing the process of bootstrapping from L to N using M as an intermediate language.
Bootstrapping to optimize a compiler

• The efficiency of programs and compilers:
  – Efficiency of programs:
    • memory usage
    • runtime
  – Efficiency of compilers:
    • Efficiency of the compiler itself
    • Efficiency of the emitted code

• Idea: Start from a simple compiler (generating inefficient code) and develop more sophisticated version of it. We can use bootstrapping to improve performance of the compiler.
Bootstrapping to optimize a compiler

- We have a host compiler of ADA to M
- Write an optimizing compiler of ADA to M in ADA
Full Bootstrapping

• A full bootstrap is necessary when building a new compiler from scratch.

• **Example:**
  • We want to implement an Ada compiler for machine \( M \). We don’t have access to any Ada compiler
  • Idea: Ada is very large, we will implement the compiler in a subset of Ada (call it Ada\(_0\)) and bootstrap it from a subset of Ada compiler in another language (e.g. C)
Full Bootstrapping (2)

- **Step 1:** build a compiler of $\text{Ada}_0$ to $M$ in another language, say $C$

- **Step 2:** compile it using a host compiler of $C$ for $M$

- **Note:** new versions would depend on the $C$ compiler for $M$
Full Bootstrapping (3)

- **Step 3:** Build another compiler of $\text{Ada}_0$ in $\text{Ada}_0$

  ![Diagram showing $\text{Ada}_0$ in $\text{Ada}_0$]

- **Step 4:** compile it using the $\text{Ada}_0$ compiler for $\text{M}$

  ![Diagram showing compilation process]

- **Note:** C compiler is no more necessary
Full Bootstrapping (4)

- **Step 5:** Build a full compiler of Ada in Ada₀

- **Step 4:** Compile it using the second Ada₀ compiler for M

- Future versions of the compiler can be written directly in Ada
Compilers
The Analysis-Synthesis Model of Compilation

• Compilers translate programs written in a language into equivalent programs in another language

• There are two parts to compilation:
  – **Analysis** determines the operations implied by the source program which are recorded in a tree structure
  – **Synthesis** takes the tree structure and translates the operations therein into the target program
Other Tools that Use the Analysis-Synthesis Model

- Editors (syntax highlighting)
- Pretty printers (e.g. Doxygen)
- Static checkers (e.g. Lint and Splint)
- Interpreters
- Text formatters (e.g. TeX and LaTeX)
- Silicon compilers (e.g. VHDL)
- Query interpreters/compilers (Databases)

Several compilation techniques are used in other kinds of systems
Compilation Phases and Passes

• Compilation of a program proceeds through a fixed series of phases
• A pass is one phase or a sequence of phases that starts from a representation of the program and produces another representation of it
• Passes can be serialized, phases not necessarily
  – Pascal, FORTRAN, C languages designed for one-pass compilation, which explains the need for function prototypes
  – Single-pass compilers need less memory to operate
  – Java and ADA are multi-pass
The Many Phases of a Compiler

Source Program

1. Lexical analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate Code Generator

Symbol-table Manager

5. Code Optimizer
6. Code Generator
7. Peephole Optimization

Error Handler

Target Program

1, 2, 3, 4: Front-End
5, 6, 7: Back-End

Analyses

Syntheses
Compiler Front- and Back-end

**Source program (character stream)**

- **Scanner (lexical analysis)**
  - Tokens
  - **Parser (syntax analysis)**
  - Parse tree
  - **Semantic Analysis and Intermediate Code Generation**
  - Abstract syntax tree or other intermediate form

**Front end analysis**

**Back end synthesis**

- **Abstract syntax tree or other intermediate form**
- **Machine-Independent Code Improvement**
- **Modified intermediate form**
- **Target Code Generation**
- **Assembly or object code**
- **Machine-Specific Code Improvement**
  - Modified assembly or object code
Scanner: Lexical Analysis

- Lexical analysis breaks up a program into tokens

```
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.
```
Context-Free Grammars

• A context-free grammar defines the syntax of a programming language
• The syntax defines the syntactic categories for language constructs
  – Statements
  – Expressions
  – Declarations
• Categories are subdivided into more detailed categories
  – A Statement is a
    • For-statement
    • If-statement
    • Assignment

\[
\text{<statement>} ::= \text{<for-statement>} | \text{<if-statement>} | \text{<assignment>}
\]
\[
\text{<for-statement>} ::= \text{for} ( \text{<expression>} ; \text{<expression>} ; \text{<expression>} ) \text{<statement>}
\]
\[
\text{<assignment>} ::= \text{<identifier>} := \text{<expression>}
\]
Example: Micro Pascal

<Program> ::= program <id> (<id> <More_ids>) ; <Block>.
<Block> ::= <Variables> begin <Stmt> <More_Stms> end
<More_ids> ::= , <id> <More_ids>
             | ε
<Variables> ::= var <id> <More_ids> : <Type> ; <More_Variables>
             | ε
<More_Variables> ::= <id> <More_ids> : <Type> ; <More_Variables>
                    | ε
<Stmt> ::= <id> := <Exp>
          | if <Exp> then <Stmt> else <Stmt>
          | while <Exp> do <Stmt>
          | begin <Stmt> <More_Stms> end
<Exp> ::= <num>
       | <id>
       | <Exp> + <Exp>
       | <Exp> - <Exp>
Parser: Syntax Analysis

- Parsing organizes tokens into a hierarchy called a **parse tree**
- Essentially, a grammar of a language defines the structure of the parse tree, which in turn describes the program structure
- A syntax error is produced by a compiler when the parse tree cannot be constructed for a program
Semantic Analysis

• Semantic analysis is applied by a compiler to discover the meaning of a program by analyzing its parse tree or abstract syntax tree.

• Static semantic checks are performed at compile time:
  – Type checking
  – Every variable is declared before used
  – Identifiers are used in appropriate contexts
  – Check subroutine call arguments
  – Check labels

• Dynamic semantic checks are performed at run time, and the compiler produces code that performs these checks:
  – Array subscript values are within bounds
  – Arithmetic errors, e.g. division by zero
  – Pointers are not dereferenced unless pointing to valid object
  – A variable is used but hasn't been initialized
  – When a check fails at run time, an exception is raised
Semantic Analysis and Strong Typing

• A language is strongly typed "if (type) errors are always detected"
  – Errors are either detected at compile time or at run time
  – Examples of such errors are listed on previous slide
  – Languages that are strongly typed are Ada, Java, ML, Haskell
  – Languages that are not strongly typed are Fortran, Pascal, C/C++, Lisp

• Strong typing makes language safe and easier to use, but potentially slower because of dynamic semantic checks

• In some languages, most (type) errors are detected late at run time which is detrimental to reliability e.g. early Basic, Lisp, Prolog, some script languages
Code Generation and Intermediate Code Forms

- A typical intermediate form of code produced by the semantic analyzer is an abstract syntax tree (AST)
- The AST is annotated with useful information such as pointers to the symbol table entry of identifiers

Example AST for the gcd program in Pascal
Target Code Generation and Optimization

• The AST with the annotated information is traversed by the compiler to generate a low-level intermediate form of code, close to assembly
• This machine-independent intermediate form is optimized
• From the machine-independent form assembly or object code is generated by the compiler
• This machine-specific code is optimized to exploit specific hardware features
Supporting Phases/Activities for Analysis

• Symbol Table Creation / Maintenance
  – Contains info (storage, type, scope, args) on each “meaningful” token, typically identifiers
  – Data structure created / initialized during lexical analysis
  – Exploited / updated during later analysis & synthesis

• Error Handling
  – Detection of different errors which correspond to all phases
  – What happens when an error is found?