1 Compilation basics

Figure 1.1 Translation.

Translation

- interpretation (transient target)
- generation (persistent target)
  - compiler: HLL -> assembly/ML

Figure 1.2 Multi-module modern compilation environment.
Object is created in modular compilation
- unresolved external references
- relocatable format
- Linker resolves external references to create relocatable load module
- Loader creates true (absolute) machine code
- not really (paging)
- Source often is pre-processed before reaching the compiler

Figure 1.3 Compilation stages. Optimization not singled out.

Figure 1.4 Tombstone representation of a program P and a translator S→T, in language L.

Figure 1.5 Representation of a machine with ML M - and execution of P on M.

Advantages of modularization and separation of machine dependency
- reusability, lower development cost and speed
Figure 1.6 Tombstone representation of translation, with two examples (host translation and cross-translation).

Figure 1.7 Two-stage compiler (cross+host).
2 Interpreters

Figure 2.1 Tombstone representation of an interpreter, with examples

Figure 2.2 Software interpreter needs to run.
3  Real and Abstract Machines

- Software interpreters provide abstract (virtual) machines

**Figure 3.1** Software interpreter for Ultima as an abstract machine.

4  Interpretive Compilers

- Hybrid of compilation and interpretation
  - lower development cost (for other languages/machines)
  - reasonable speed
  - Pascal 2-stage compiler is a good example
    - compile source into machine independent low-level representation P-code
    - interpret P-code through machine dependent interpreter
    - trades some efficiency for greater portability

**Figure 4.1** Pascal interpretive compiler.
5 Portable Compilers

- For portability, compilers are
  - developed in exchangable modules
  - implemented in HLL

Figure 5.1 Suppose we have the three high level elements. Can we get execution? No unless we have an interpreter at least.
Figure 5.2 We implement P-code -> C interpreter in C and compile it.

Figure 5.3 Now Pascal programs can be compiled with the interpreter into P-code and then executed through the same interpreter.

Figure 5.4 New 2-stage Pascal generative compiler. Inefficient compilation. Can be improved through bootstrapping.
6 Bootstrapping

- a number of techniques which rely on partial/deficient compiler version to create a full/better version
- often compiling a translator expressed in its own language

Figure 6.1 Bootstrapping a portable compiler. Here we attempt to replace the interpreter in the Pascal translation environment with a generator, and then we improve efficiency of both generators in the 2-stage setup by compiling the generators.
Figure 6.2 More efficient 2-stage Pascal generative compiler.

- Full bootstrap
  - implementing the translator in a subset of the same language, then recompiling

Figure 6.3 Full bootstrap of Ada. Start with a subset ADA-S. If you cannot imagine why do this, imagine Ada is the first Fortran. There is no HLL to implement the compiler.
Half bootstrap

- when moving a translator implemented in itself to a new host

**Figure 6.4** We have Pascal translator, and its executable, on machine HM. Move to TM.
Bootstrapping can be used to improve efficiency.

**Figure 6.5** Suppose we have Pascal compiler in M (X for inefficient version). Improve efficiency at both translation and execution of targets.