CS342: Organization of Prog. Languages

Topic 5: Interpreters and Transformers

- Compilers and Interpreters

- Transformers

- Anatomy of a Compiler

- A notation for interpreters

- A notation for transformers

- Cross-Compilation and Boot-strapping

Compilers and Interpreters

- Compilers and Interpreters are often portrayed as two alternatives to language implementation. *This is a false dichotomy!*

- A more accurate view is:
  An "interpreter" takes a representation of a program and executes it.
  A "transformer" takes some representation of a program and creates an equivalent program.

- An interpreter may be implemented in software, hardware, or some combination.

- A transformer may take a representation of the program in one language and produce a representation in another. This is a "translator."

- A translator may do sophisticated whole-program analysis and produce code in a lower-level language. This a "compiler."

- A translator may use the same language for its input and output, but improve the efficiency of the code. This is an "optimizer."

Interpreters

- Perform localized change of program representation

- Execute program as represented

- View program as a request for services

- E.g. Byte-code interpreter in JVM

- E.g. PowerPC instruction interpreter in hardware

Transformers

- Take program as input, produce equivalent program as output.

- Can perform global "program-as-a-whole" analysis, translation and reorganization

- Translators, e.g.
  Modula-3 -> C
  Macsyma -> Lisp
  Fortran -> i686

- Optimizers

- Compilers, typically translators + optimizes

Compilation Phases − Simplified

- Lexical Analysis

- Syntactic Analysis

- Semantic Analysis

- Intermediate code generation

- Intermediate code improvement

- Machine-specific code generation

- Machine-specific code improvement
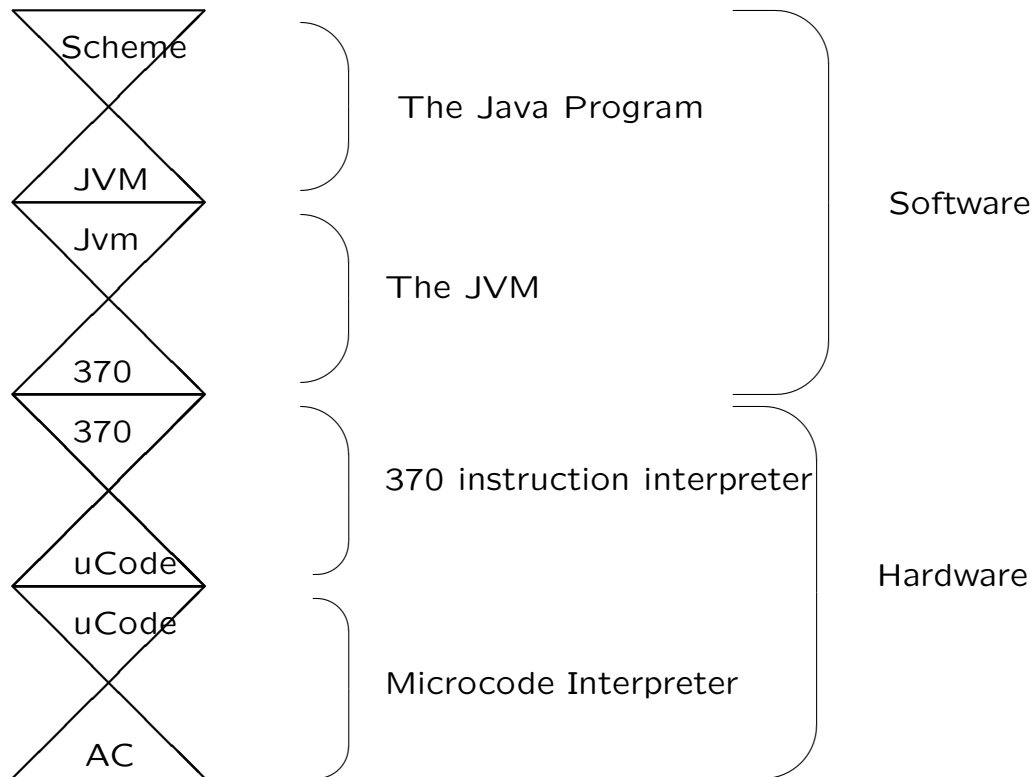
A Notation for Interpreters

- We use the following diagram as the notation for an interpreter:

```
\   H   /
 \     /
  \   /
   \ /
   / \
  /   \
 /  X  \
/       \
```
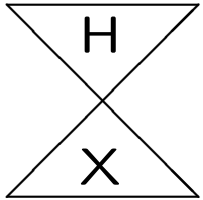
"H" is the language being interpreted

"X" is the language in which the interpreter is implemented
(i.e. the language of its executable program)

- E.g. A Scheme interpreter written in Java and compiled to JVM running on an IBM mainframe can be seen as a composition of 4 interpreters, some in software, some in hardware.

A Notation for Transformers

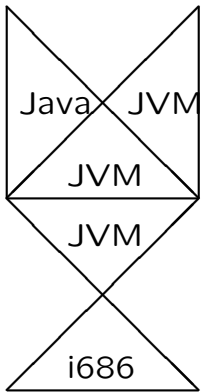- We use the following diagram as the notation for a transformer:

$$\begin{array}{c} \text{I} \quad \boxtimes \quad \text{O} \\ \text{X} \end{array}$$
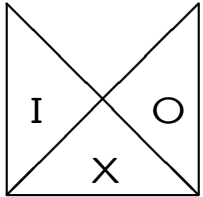
  "I" is the language of the input program
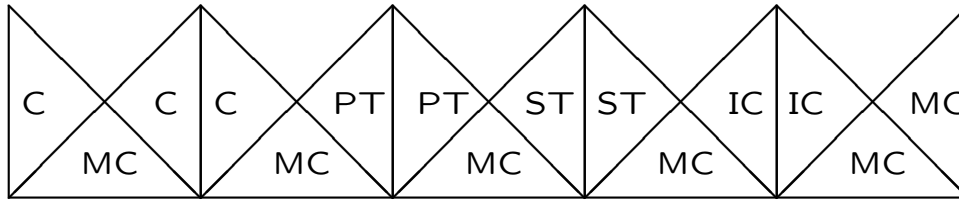
  "O" is the language of the output program

  "X" is the language in which the transforming program is imple-mented

- E.g. A Java compiler for a PC

- E.g. A C Compiler

```
C       C  C       PT  PT  ST  ST     IC  IC     MC
    MC          MC          MC          MC          MC
```

The steps are
- C preprocessor: takes C source code to C source code, handling `#include` etc.

- Parser: takes C source code to parse trees (PT).

- Semantic analysis: Takes parse trees to semantic trees (ST), e.g. with type info.

- Intermediate code generation: Takes semantic trees to a machine–independent intermediate code (IC).

- Intermediate code optimization (not shown): takes IC to IC.

- Target machine code generation: takes IC to target machine code (MC).

- Target machine code optimization (not shown): takes MC to MC.

- Sometimes a single program can both translate and interpret.

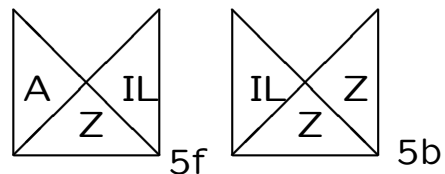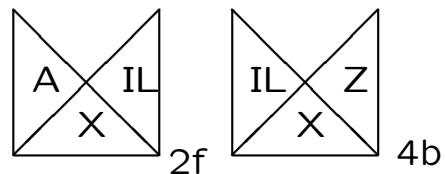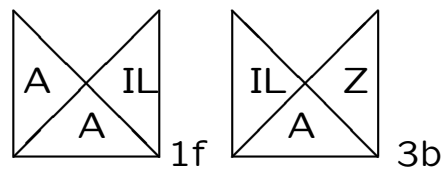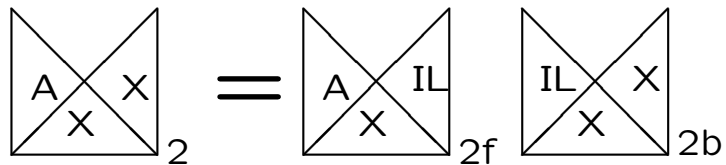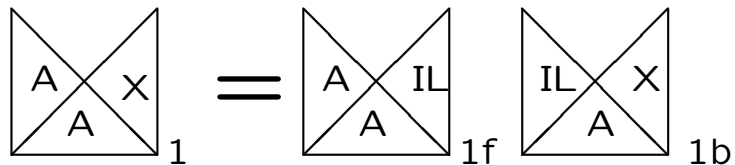
Scheme
Interperter/Compiler
Source
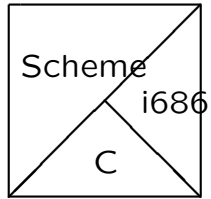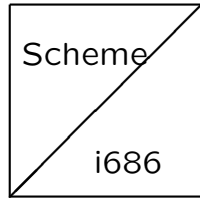
Scheme
Interpreter/Compiler
Object

We see this

– in integrated development environments (IDEs)

– when dynamic compilation is used

– in optimizers.

Cross-Compilation and Boot-strapping

- Suppose we have a compiler for language A running on some machine X and we wish to construct a compiler which runs on machine Z.

$$\boxed{\begin{smallmatrix}A & X \\ & A\end{smallmatrix}}_1 = \boxed{\begin{smallmatrix}A & IL \\ & A\end{smallmatrix}}_{1f} \boxed{\begin{smallmatrix}IL & X \\ & A\end{smallmatrix}}_{1b}$$

$$\boxed{\begin{smallmatrix}A & X \\ & X\end{smallmatrix}}_2 = \boxed{\begin{smallmatrix}A & IL \\ & X\end{smallmatrix}}_{2f} \boxed{\begin{smallmatrix}IL & X \\ & X\end{smallmatrix}}_{2b}$$

$$\boxed{\begin{smallmatrix}A & IL \\ & A\end{smallmatrix}}_{1f} \boxed{\begin{smallmatrix}IL & Z \\ & A\end{smallmatrix}}_{3b}$$

$$\boxed{\begin{smallmatrix}A & IL \\ & X\end{smallmatrix}}_{2f} \boxed{\begin{smallmatrix}IL & Z \\ & X\end{smallmatrix}}_{4b}$$

$$\boxed{\begin{smallmatrix}A & IL \\ & Z\end{smallmatrix}}_{5f} \boxed{\begin{smallmatrix}IL & Z \\ & Z\end{smallmatrix}}_{5b}$$

8

Step 1 Have the source code (1) and executable program (2) for a compiler for language "A" on the machine "X"

Note that the compiler is composed of many phases, the first group produce a machine-independent form of the compiled program in an intermediate language (IL), and the second group takes this IL program to machine-specific code for "X"

We group these as (1f)+(1b) and (2f) + (2b).

Step 2 Write an `IL -> Z code` generator (3b) in source language A.

This is the main job. It might be accomplished by adapting the program (1b).

Compile the program (3b) on machine X using compiler (2) to get the executable program (4b).

Step 3 Now (3b) together with (1f) form the source for a full compiler for machine Z.

And (2f) + (4b) provide an executable compiler running on machine X to produce code for machine Z.

We now have a "cross-compiler" for machine Z running on X.

Step 4 Use the cross-compiler (2f)+(4b) to recompile the source programs (1f)+(3b) to get the executable programs (5f)+(5b) which run on machine Z. We are done!