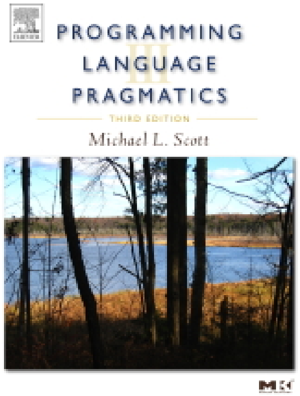
Introduction

UNIVERSIDADE DA CORUÑA

GRAO EN ENXEÑERÍA INFORMÁTICA

DESEÑO DAS LINGUAXES DE PROGRAMACIÓN

Based on Chapter 1 of:
Michael L. Scott, *Programming Language Pragmatics*,
Morgan Kaufmann, 2008.



Introduction

- Why are there so many programming languages?
  - evolution -- we've learned better ways of doing things over time
  - socio-economic factors: proprietary interests, commercial advantage
  - orientation toward special purposes
  - orientation toward special hardware
  - diverse ideas about what is pleasant to use

Introduction

- What makes a language successful?
  - easy to learn (BASIC, Pascal, LOGO, Scheme, Scratch)
  - easy to express things, easy use once fluent, "powerful" (C, Common Lisp, APL, Algol-68, Perl)
  - easy to implement (BASIC, Forth)
  - possible to compile to very good (fast/small) code (Fortran)
  - backing of a powerful sponsor (COBOL, PL/1, Ada, Visual Basic)
  - wide dissemination at minimal cost (Pascal, Turing, Java)

Introduction

- Why do we have programming languages? What is a language for?
  - way of thinking -- way of expressing algorithms
  - languages from the user's point of view
  - abstraction of virtual machine -- way of specifying what you want
  - the hardware to do without getting down into the bits
  - languages from the implementor's point of view

Why study programming languages?

- Help you choose a language.
  - C vs. Modula-3 vs. C++ for systems programming
  - Fortran vs. APL vs. Ada for numerical computations
  - Ada vs. Modula-2 for embedded systems
  - Common Lisp vs. Scheme vs. ML for symbolic data manipulation
  - Java vs. C/CORBA for networked PC programs

Why study programming languages?

- Make it easier to learn new languages some languages are similar; easy to walk down family tree
  - concepts have even more similarity; if you think in terms of iteration, recursion, abstraction (for example), you will find it easier to assimilate the syntax and semantic details of a new language than if you try to pick it up in a vacuum. Think of an analogy to human languages: good grasp of grammar makes it easier to pick up new languages (at least Indo-European).

Why study programming languages?

- Help you make better use of whatever language you use
  - understand obscure features:
    - In C, help you understand unions, arrays & pointers, separate compilation, varargs, catch and throw
    - In Common Lisp, help you understand first-class functions/closures, streams, catch and throw, symbol internals

Why study programming languages?

- Help you make better use of whatever language you use (2)
  - understand implementation costs: choose between alternative ways of doing things, based on knowledge of what will be done underneath:
    - use simple arithmetic equal (use x*x instead of x**2)
    - use C pointers or Pascal "with" statement to factor address calculations
    - avoid call by value with large data items in Pascal
    - avoid the use of call by name in Algol 60
    - choose between computation and table lookup (e.g. for cardinality operator in C or C++)

Why study programming languages?

- Help you make better use of whatever language you use (3)
  - figure out how to do things in languages that don't support them explicitly:
    - lack of suitable control structures in Fortran
    - use comments and programmer discipline for control structures
    - lack of recursion in Fortran, CSP, etc
    - write a recursive algorithm then use mechanical recursion elimination (even for things that aren't quite tail recursive)

Why study programming languages?

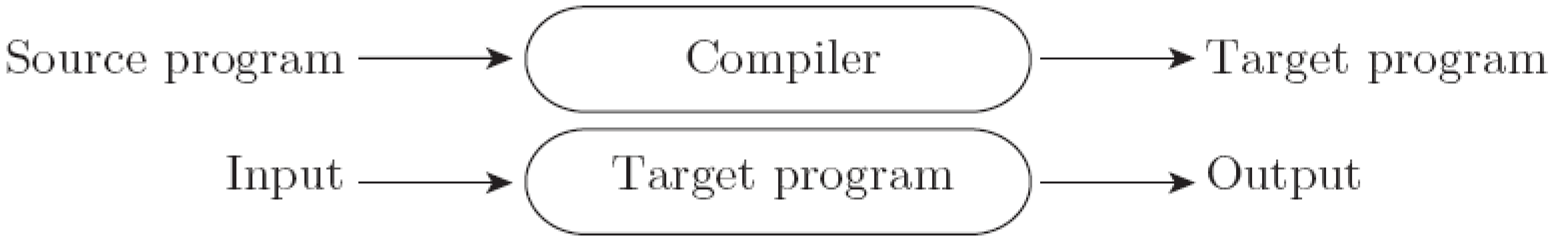
- Help you make better use of whatever language you use (4)
  - figure out how to do things in languages that don't support them explicitly:
    - lack of named constants and enumerations in Fortran
    - use variables that are initialized once, then never changed
    - lack of modules in C and Pascal use comments and programmer discipline
    - lack of iterators in just about everything fake them with (member?) functions

Imperative languages

- Group languages as
  - imperative
    - von Neumann (Fortran, Pascal, Basic, C)
    - object-oriented (Smalltalk, Eiffel, C++?)
    - scripting languages (Perl, Python, JavaScript, PHP)
  - declarative
    - functional (Scheme, ML, pure Lisp, FP)
    - logic, constraint-based (Prolog, VisiCalc, RPG)

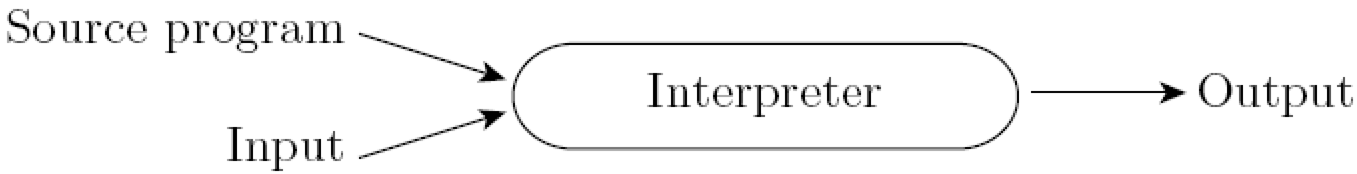
Compilation vs. Interpretation

- Compilation vs. interpretation
  - not opposites
  - not a clear-cut distinction
- Pure Compilation
  - The compiler translates the high-level source program into an equivalent target program (typically in machine language), and then goes away:



Compilation vs. Interpretation

- Pure Interpretation
  - Interpreter stays around for the execution of the program
  - Interpreter is the locus of control during execution



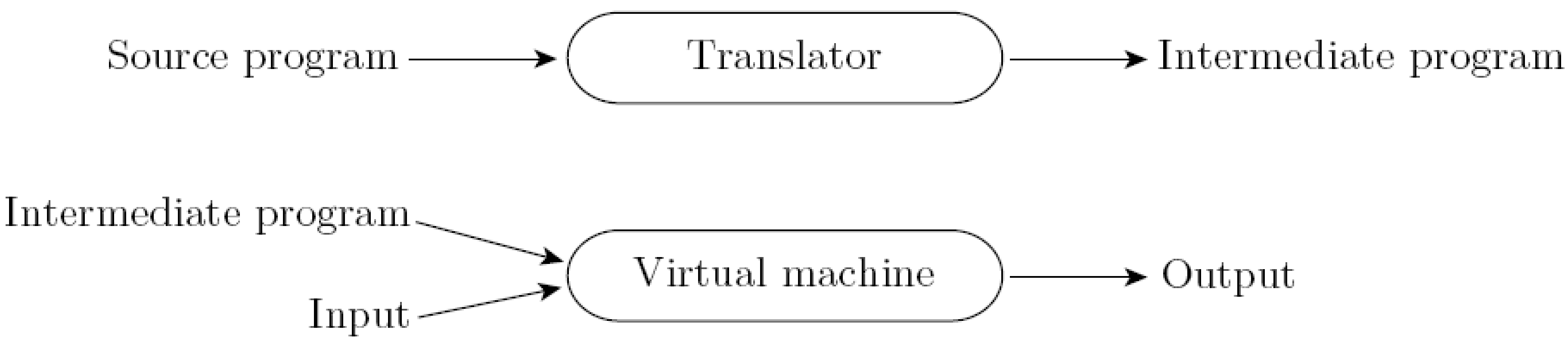
Compilation vs. Interpretation

- Interpretation:
  - Greater flexibility
  - Better diagnostics (error messages)

- Compilation
  - Better performance

Compilation vs. Interpretation

- Common case is compilation or simple pre-processing, followed by interpretation
- Most language implementations include a mixture of both compilation and interpretation



Compilation vs. Interpretation

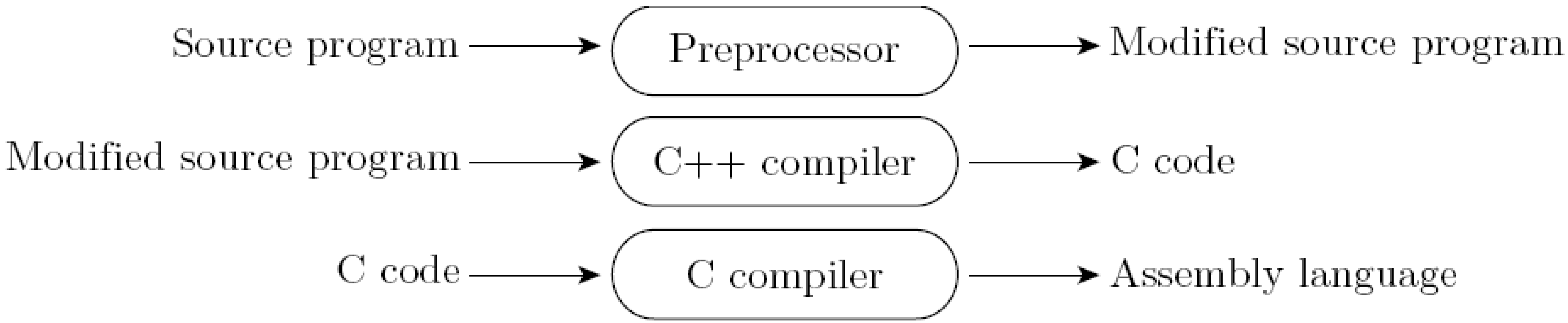
- Note that compilation does NOT have to produce machine language for some sort of hardware
- Compilation is *translation* from one language into another, with full analysis of the meaning of the input
- Compilation entails semantic *understanding* of what is being processed; pre-processing does not
- A pre-processor will often let errors through. A compiler hides further steps; a pre-processor does not

Compilation vs. Interpretation

- Many compiled languages have interpreted pieces, e.g., formats in Fortran or C
- Most use “virtual instructions”
  - set operations in Pascal
  - string manipulation in Basic
- Some compilers produce nothing but virtual instructions, e.g., Pascal P-code, Java byte code, Microsoft COM+

Compilation vs. Interpretation

- Implementation strategies:
  - Source-to-Source Translation (C++)
    - C++ implementations based on the early AT&T compiler generated an intermediate program in C, instead of an assembly language:



Compilation vs. Interpretation

- Implementation strategies:
  - Compilation of Interpreted Languages
    - The compiler generates code that makes assumptions about decisions that won’t be finalized until runtime. If these assumptions are valid, the code runs very fast. If not, a dynamic check will revert to the interpreter.

Compilation vs. Interpretation

- Implementation strategies:
  - Dynamic and Just-in-Time Compilation
    - In some cases a programming system may deliberately delay compilation until the last possible moment.
      - Lisp or Prolog invoke the compiler on the fly, to translate newly created source into machine language, or to optimize the code for a particular input set.
      - The Java language definition defines a machine-independent intermediate form known as *byte code*. Byte code is the standard format for distribution of Java programs.
      - The main C# compiler produces .NET Common Intermediate Language (CIL), which is then translated into machine code immediately prior to execution.

Compilation vs. Interpretation

- Compilers exist for some interpreted languages, but they aren't pure:
  - selective compilation of compilable pieces and extra-sophisticated pre-processing of remaining source.
  - Interpretation of parts of code, at least, is still necessary for reasons above.
- Unconventional compilers
  - text formatters
  - silicon compilers
  - query language processors

Programming Environment Tools

- Tools

Type	Unix examples
Editors	vi, emacs
Pretty printers	cb, indent
Pre-processors (esp. macros)	cpp, m4, watfor
Debuggers	adb, sdb, dbx, gdb
Style checkers	lint, purify
Module management	make
Version management	sccs, rcs
Assemblers	as
Link editors, loaders	Id, Id-so
Perusal tools	More, less, od, nm
Program cross-reference	ctags