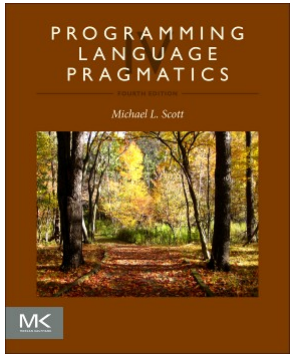
CMPT 341: Programming Languages

Spring 2020

Chapter 1 :: Introduction

Programming Language Pragmatics, Fourth Edition

Michael L. Scott



ELSEVIER

Introduction

- Programmers used to write programs in machine language in 1940s.

```
55 89 e5 53  83 ec 04 83  e4 f0 e8 31  00 00 00 89  c3 e8 2a 00
00 00 39 c3  74 10 8d b6  00 00 00 00  39 c3 7e 13  29 c3 39 c3
75 f6 89 1c  24 e8 6e 00  00 00 8b 5d  fc c9 c3 29  d8 eb eb 90
```

GCD program in x86 machine language

- Very hard to read & write programs.

Introduction

```
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $4, %esp
        andl    $-16, %esp
        call    getint
        movl    %eax, %ebx
        call    getint
        cmpl    %eax, %ebx
        je      C
A:      cmpl    %eax, %ebx
        jle     D
        subl    %eax, %ebx
B:      cmpl    %eax, %ebx
        jne     A
C:      movl    %ebx, (%esp)
        call    putint
        movl    -4(%ebp), %ebx
        leave
        ret
D:      subl    %ebx, %eax
        jmp     B
```

GCD program in x86 assembler

- Assembly language:
  - More readable version of machine language.
  - One to one correspondence to machine language.

Introduction

- How was an assembler written?
  - First assembler handwritten in machine language (basic version).
  - Next updated versions? Written in assembly language.

Introduction

- As programs becomes more sophisticated:
  - Writing in assembly language became unfeasible.
  - Rewriting programs for every new machine is frustrating.

Introduction

- Programmers needed:
  - Express/write programs in a way to resemble mathematical formulae (at that time).
  - Machine-independent language.
- Mid 1950s, Fortran was developed and sooner followed by Lisp and Algol.

Introduction

- Why are there so many programming languages?
  - evolution -- we've learned better ways of doing things over time (goto in Fortran, Cobol, etc.)
  - 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)
  - 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. C++ vs. C# for systems programming
  - Fortran vs. C for numerical computations
  - PHP vs. Ruby for web-based applications
  - Ada vs. C for embedded systems
  - Common Lisp vs. Scheme vs. ML for symbolic data manipulation
  - Java vs. .NET 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.
  - 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)

Imperative languages

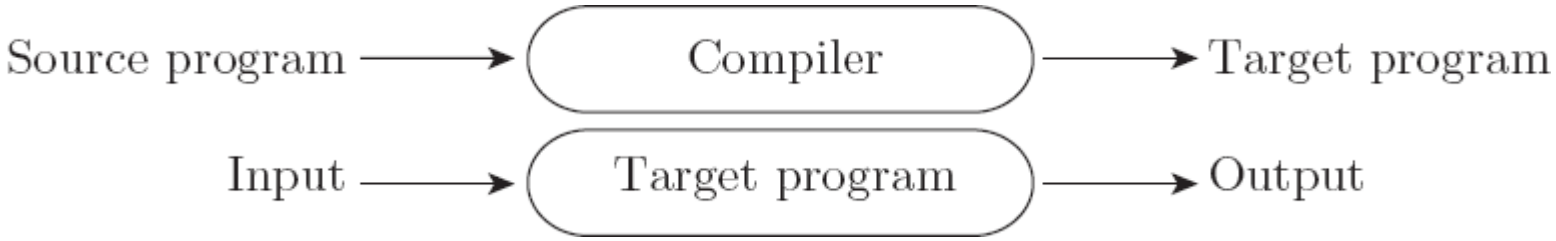
- Group languages as
  - imperative
    - von Neumann (Fortran, Pascal, Basic, C)
    - object-oriented (Smalltalk, Eiffel, C++?)
    - scripting languages (Perl, Python, JavaScript, PHP)
  - declarative
    - functional (Scheme, ML, pure Lisp, FP, Scala)
    - logic, constraint-based (Prolog, VisiCalc,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

Imperative languages

- Imperative languages, particularly the von Neumann languages, predominate
  - They will occupy the bulk of our attention
- We also plan to spend some time on functional, logic languages

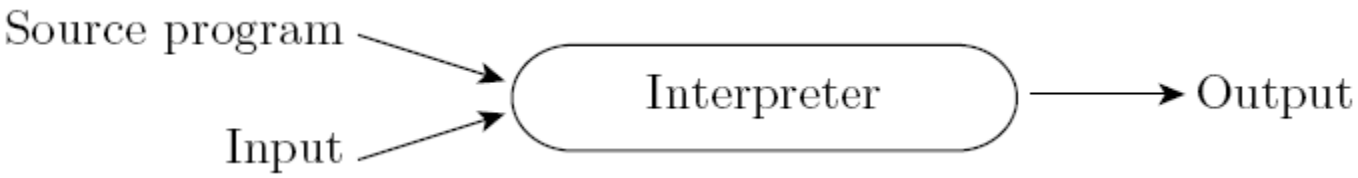
Compilation vs. Interpretation

- 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

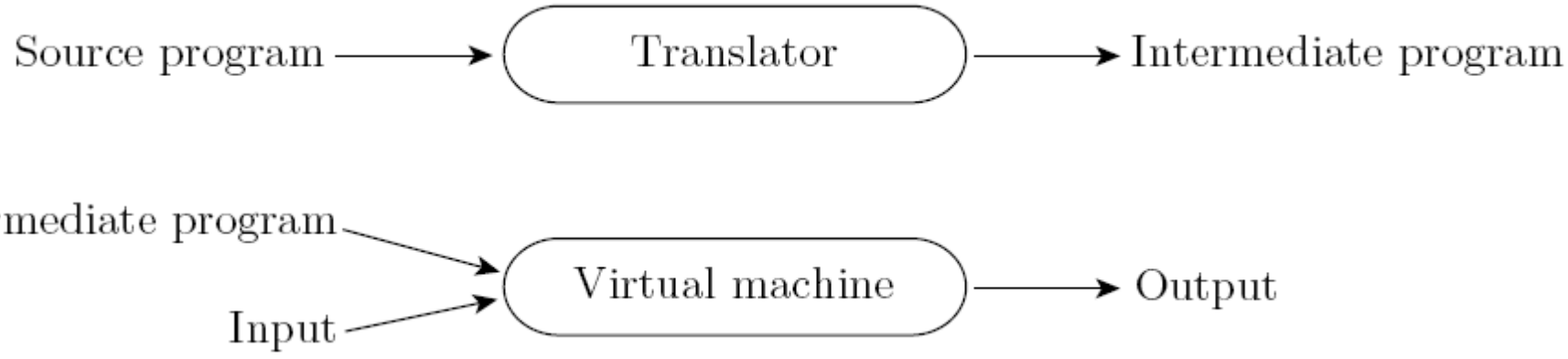


Compilation vs. Interpretation

- 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

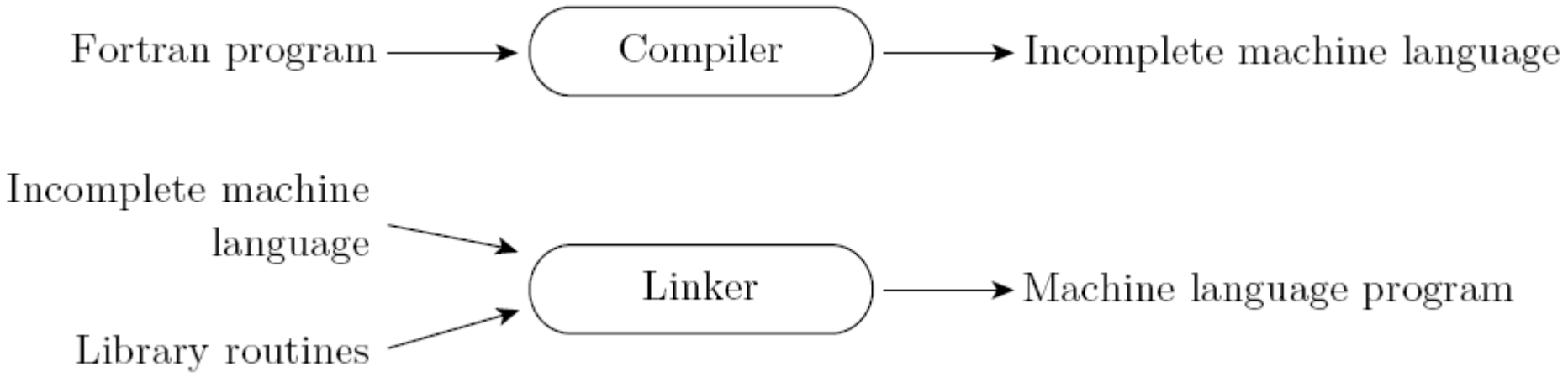
- Many compiled languages have interpreted pieces, e.g., formats in Fortran or C
- Most use "virtual instructions"
  - set operations in Pascal
  - string manipulation in Basic
- Some compilers produce nothing but virtual instructions, e.g., Pascal P-code, Java byte code, Microsoft COM+

Compilation vs. Interpretation

- Implementation strategies:
  - Preprocessor
    - Removes comments and white space
    - Groups characters into *tokens* (keywords, identifiers, numbers, symbols)
    - Expands abbreviations in the style of a macro assembler
    - Identifies higher-level syntactic structures (loops, subroutines)

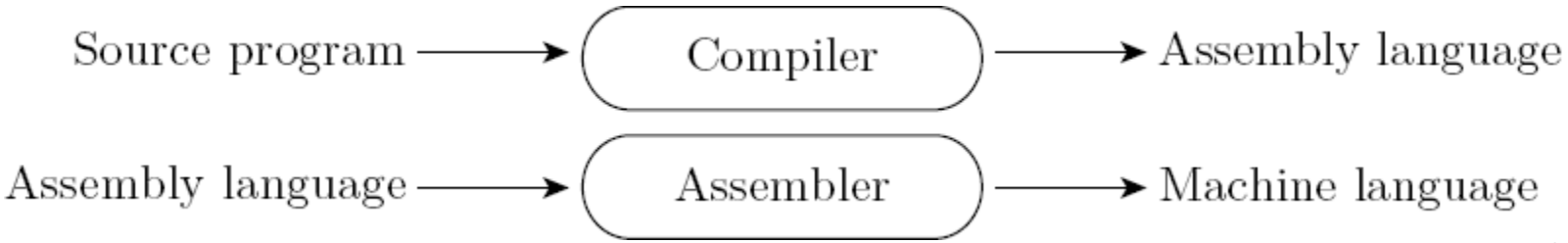
Compilation vs. Interpretation

- Implementation strategies:
  - Library of Routines and Linking
    - Compiler uses a *linker* program to merge the appropriate *library* of subroutines (e.g., math functions such as sin, cos, log, etc.) into the final program:



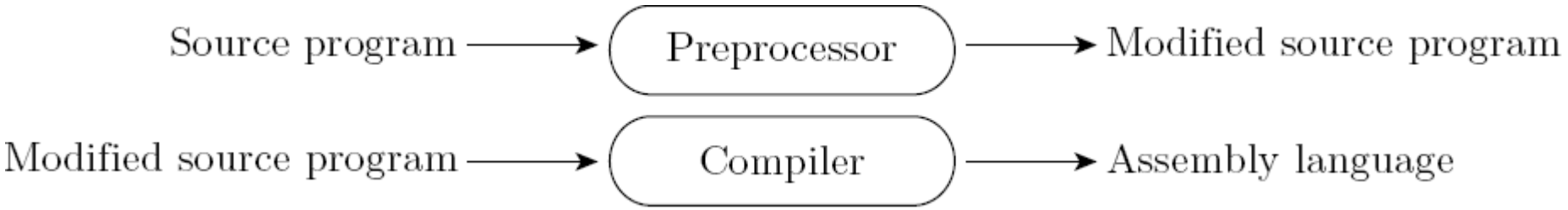
Compilation vs. Interpretation

- Implementation strategies:
  - Post-compilation Assembly
    - Facilitates debugging (assembly language easier for people to read)
    - Isolates the compiler from changes in the format of machine language files (only assembler must be changed, is shared by many compilers)



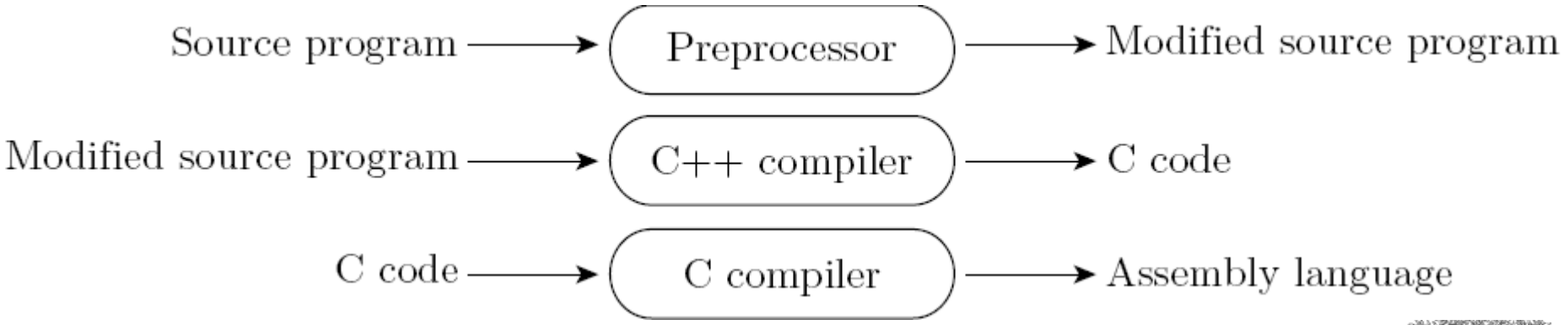
Compilation vs. Interpretation

- Implementation strategies:
  - The C Preprocessor (conditional compilation)
    - Preprocessor deletes portions of code, which allows several versions of a program to be built from the same source



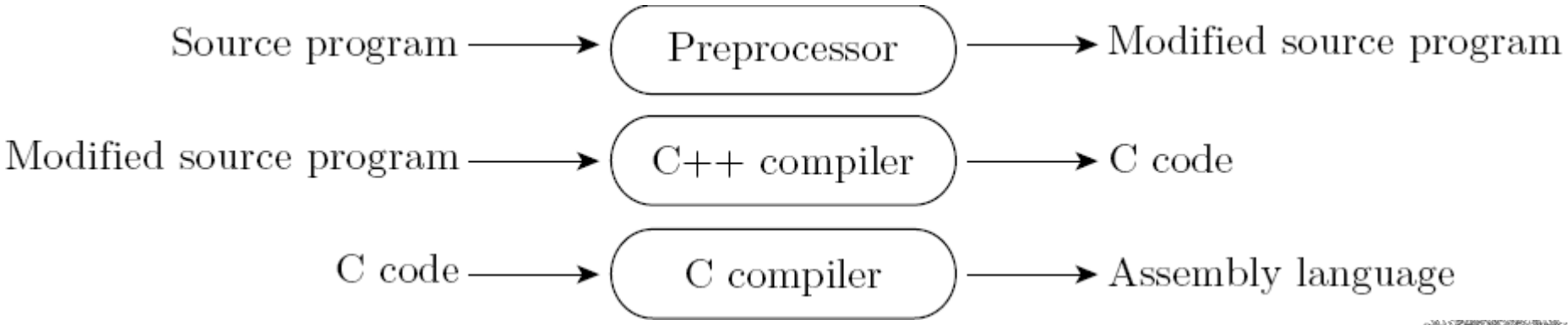
Compilation vs. Interpretation

- 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:



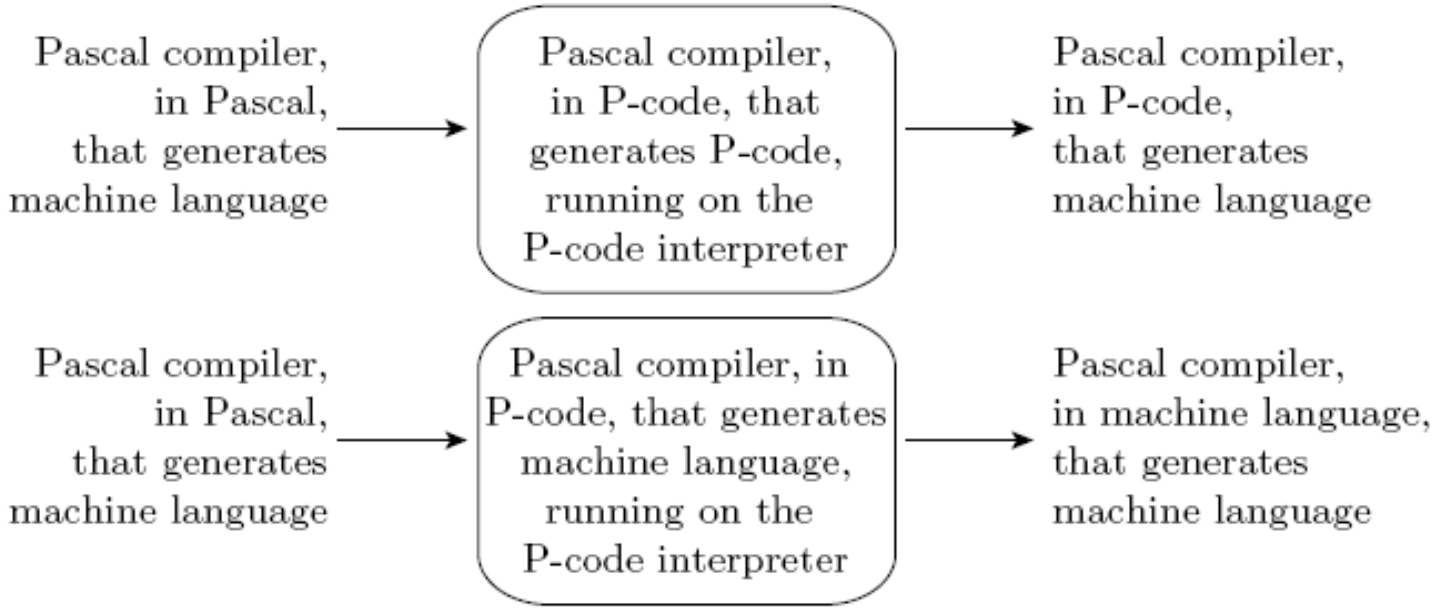
Compilation vs. Interpretation

- Implementation strategies:
  - 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:
  - *Bootstrapping*



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

- Implementation strategies:
  - Microcode (on machines designed before mid-1980s)
    - Assembly-level instruction set is not implemented in hardware; it runs on an interpreter.
    - Interpreter is written in low-level instructions (*microcode* or *firmware*), which are stored in read-only memory and executed by the hardware.

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 (TEX)
  - silicon compilers (compiler to integrated circuits (IC))
  - query language processors (SQL)

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	`ld`, `ld-so`
Perusal tools	`More`, `less`, `od`, `nm`
Program cross-reference	`ctags`