Chapter 1, Introduction to Programming Languages

Programming Languages

Objective

Chapter’s primary lesson:

- Provide an overview of translating code into machine language and interpreting code
- Provide an overview of the compilation process

Assemblers versus Compilers

Assemblers are much simpler than compilers with an almost 1-1 correspondence between assembly instructions and machine instructions.

Compilers do a much more complicated translation.

Compilers versus Interpreters

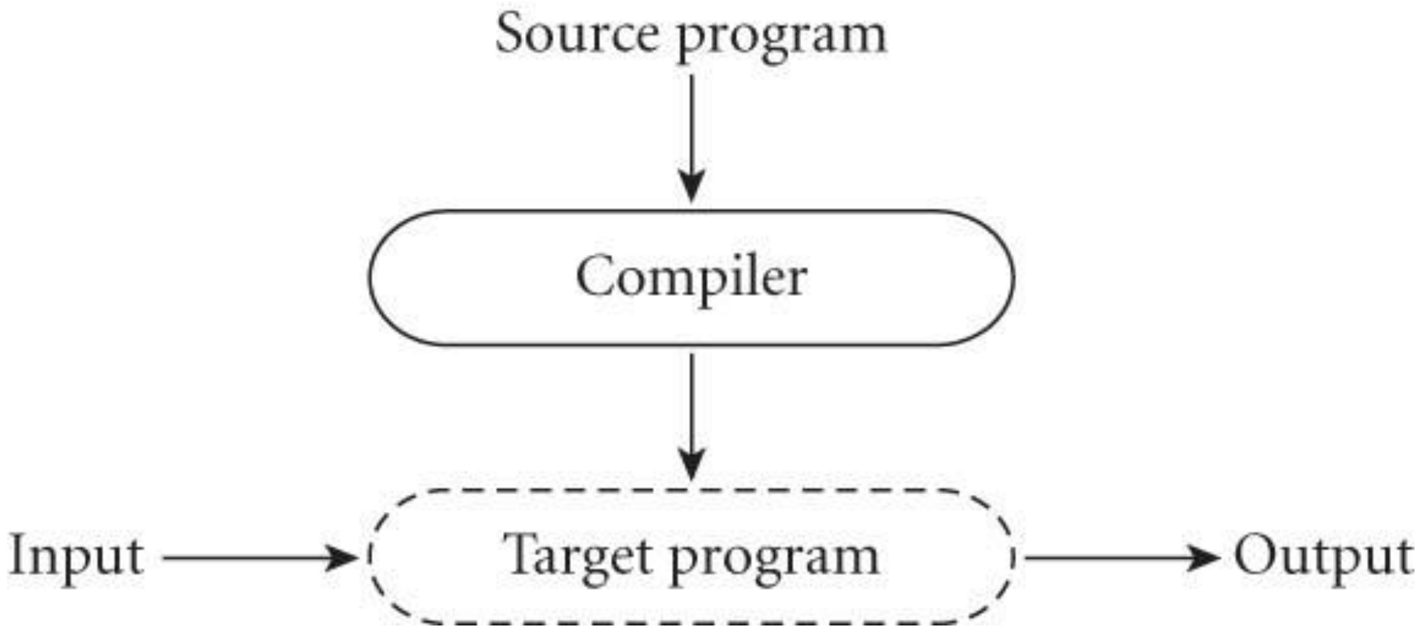
Compilers :

- Thorough job of analysis
- Results appears substantially different than the source

Interpreters :

- Remains the locus of control, they stay around for the execution of the application
- Reads statements more or less one at a time, executing them as it goes

Pure compilation



Source program

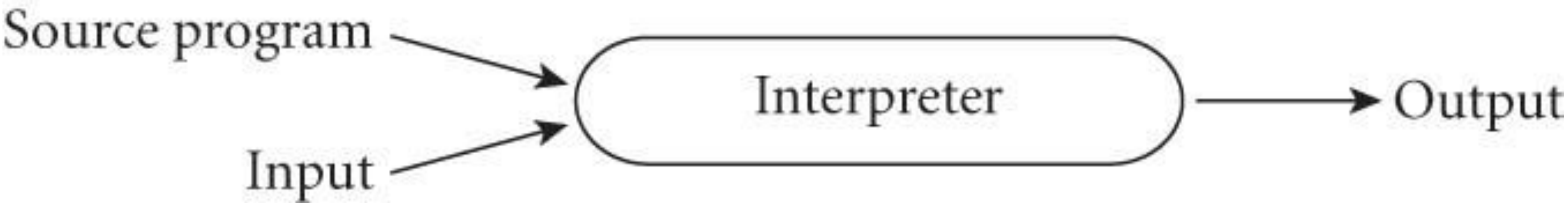
Compiler

Input

Target program

Output

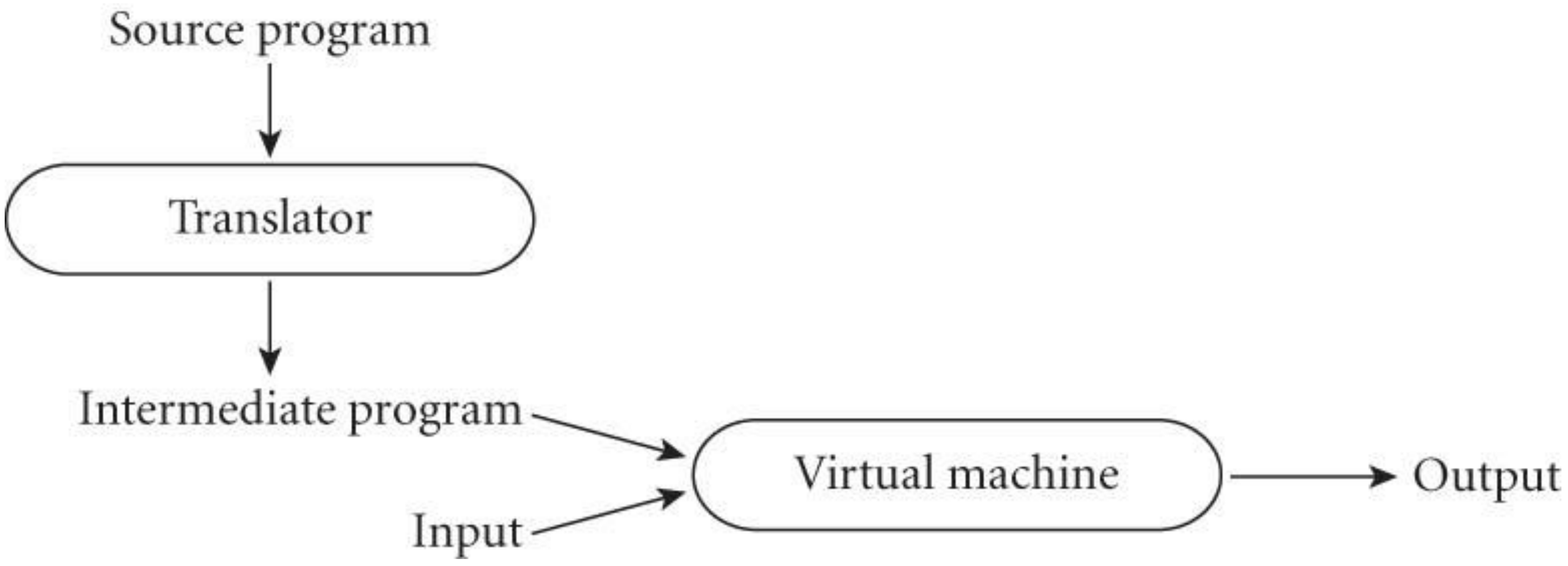
Pure Interpretation



Compilation versus Interpretation

Compilation	Interpretation
Generally execute faster	Overhead of translating at run time makes interpreted languages slower
Typically strongly typed so more robust	Often loosely typed so programs are quicker to write, are more flexible, but are less robust
Errors can be found during compilation process making the program safer	Errors found while interpreting, so diagnostics can be better but program may be less safe

Most Languages mix compilation and interpretation



Virtual machine - stays around like an interpreter , complicated interpreter that executes an intermediate language

Preprocessing

Preprocessing relates to both compilation and interpretation.

Preprocessing associated with compilation – may remove comments, expand macros, include libraries, create constants (define),etc.

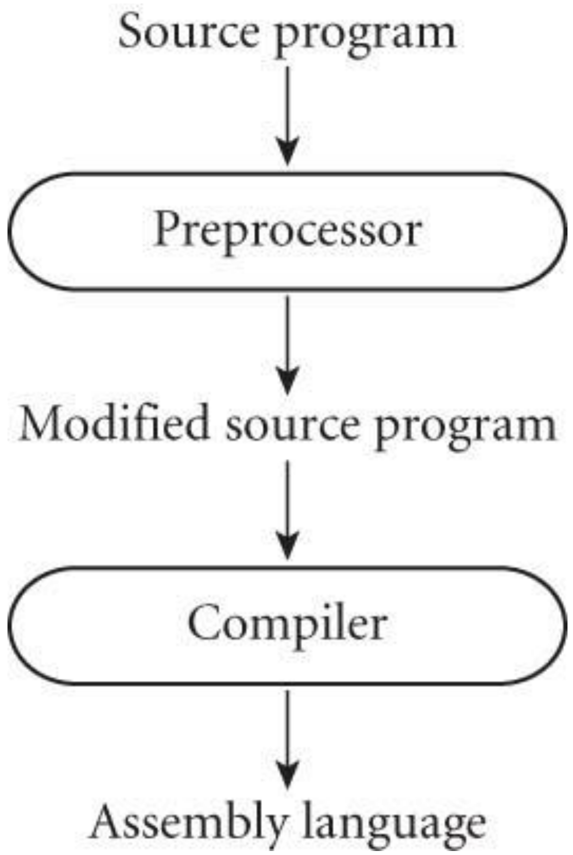
Preprocessing associated with interpreters may remove comments and white space, and group characters together into tokens such as keywords, identifiers, numbers, and symbols.

Dynamic and Just-In-Time Compilation

- Java -------> byte code Send over Internet to run on any
  complied platform
  (Could use just-in-time compiler
  byte code ---> machine code)

- C# ---------> CIL Send over Internet to run on certain
  compiled platform
  (Uses just-in-time compiler
  CIL ----> machine code)

The C Preprocessor



Source program

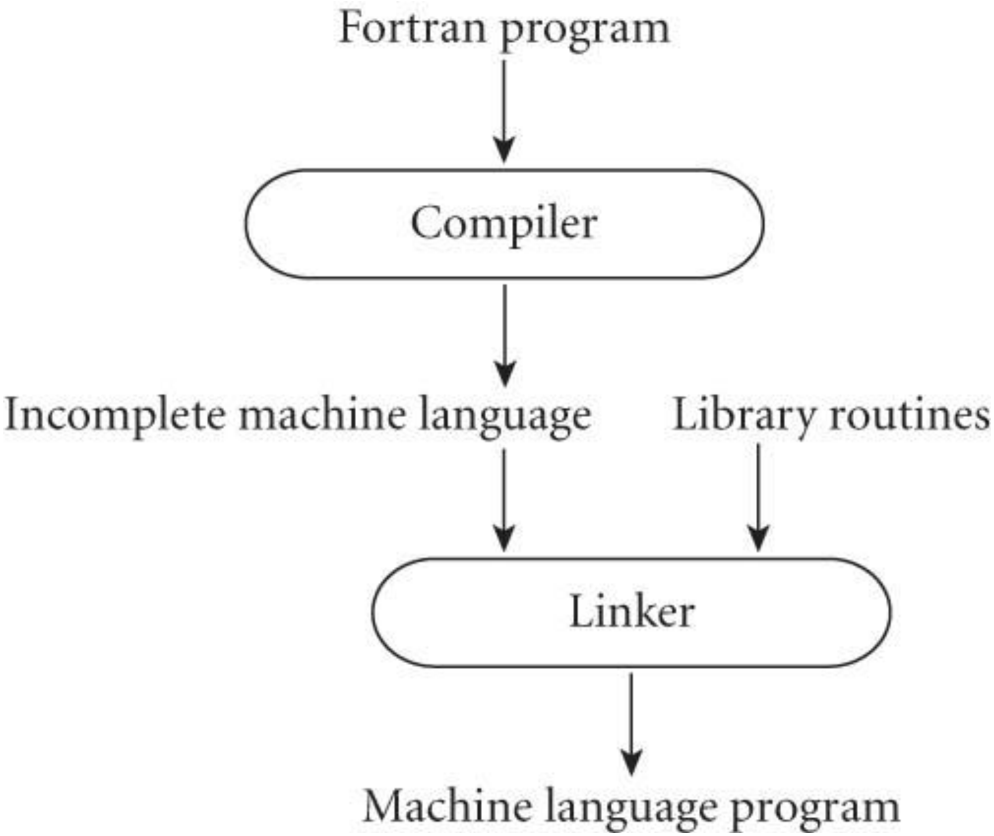
Preprocessor

Modified source program

Compiler

Assembly language

Library Routines and Linking (compilation)



Fortran program

Compiler

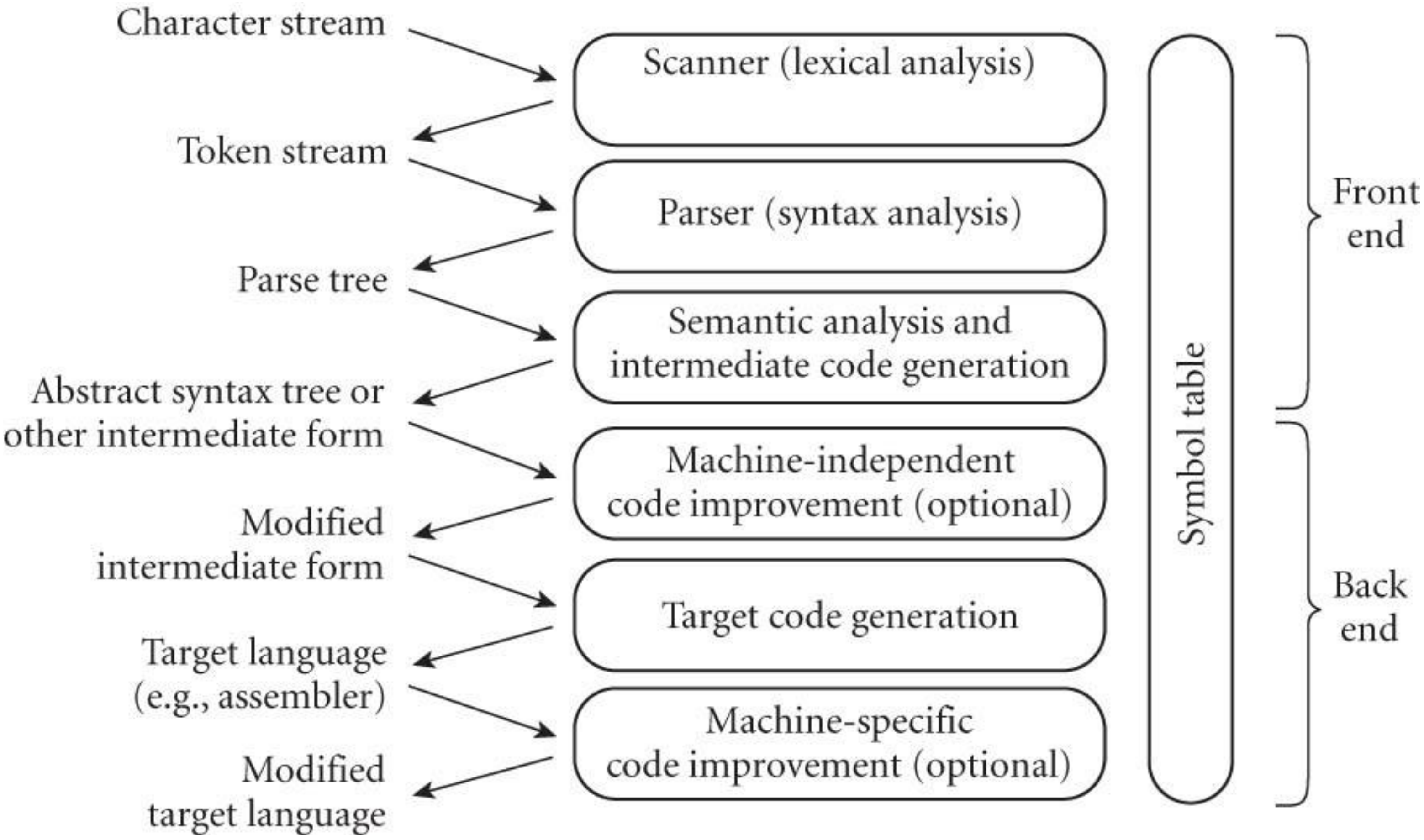
Incomplete machine language

Library routines

Linker

Machine language program

Phases of Compilation


Character stream
Token stream
Parse tree
Abstract syntax tree or other intermediate form
Modified intermediate form
Target language (e.g., assembler)
Modified target language
Scanner (lexical analysis)
Parser (syntax analysis)
Semantic analysis and intermediate code generation
Machine-independent code improvement (optional)
Target code generation
Machine-specific code improvement (optional)
Symbol table
Front end
Back end

Lexical Analysis

Scanning is also known as lexical analysis

Scanners group symbols into tokens based on rules expressed as regular expressions

Scanners also:

- Remove comments
- Remove extraneous characters like white space
- Saves text of ids, strings, numeric literals
- Tags tokens with line and column number so errors can be more informative

Parser

Parser uses grammar rules (from a context-free grammar) to build a parse tree

```
loop {
        request a token from the scanner
        builds parse tree
  }
```

Static Semantic Analysis

Semantic analysis builds a symbol table and captures errors that couldn't be caught by just building the parse tree.

Static semantic errors caught by C compliers (page 29)

- Undeclared identifier (variable for example)
- Identifiers in inappropriate context (calling an integer as a subroutine, adding a string to an integer, etc.)
- Subroutine call providing the wrong number or type of arguments
- Label on the arms of a switch statement are repeated or aren't constants
- Non-void functions not explicitly returning a value

Dynamic Semantic Analysis

Dynamic semantic errors aren't caught until run time. These are errors that couldn't be caught statically. Instead the compiler generated code to catch these. Developers of C opted to not catch many of these.

Dynamic semantic errors caught by languages other than C: (page 29)

- Unassigned variables used in an expression
- Dangling pointers dereferenced
- Out of bound array subscripts
- Overflow or arithmetic operation

Other Errors

Not counting simple program bugs, there are still errors caused by using the programming language incorrectly which the compiler can neither catch nor easily generated code to catch.

Compilation and Catching Errors

Times when errors can be caught:

- syntax error detected by the lexical analyzer
- syntax error detected by the parser
- static semantic analysis error
- dynamic semantic analysis error
- compiler can't catch, may be caught by the hardware