CS 403: Principles of Programming Languages

Stefan D. Bruda

Fall 2024

- Several subjects:
  - An introduction to functional programming using Haskell
  - An introduction to logic programming using Prolog
  - Formal description of programming languages
  - The compilation process (recursive descent)
  - A more in-depth look at the procedural paradigm

Why are there so many programming languages?

- Evolution = we've learned better ways of doin 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
- Hardware limitations (historical)

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, Python)
- Easy to implement (C, 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)

Why do we have programming languages?

- Because writing machine code is painful

What is a language for?

- Way of thinking $\rightarrow$ way of expressing algorithms
  - Languages from the user's point of view
- Abstraction of virtual machine $\rightarrow$ way of specifying what you want
  - Tell the hardware what to do without getting down to bits
  - Languages from the implementor's point of view

Why study programming languages?

- Make it easier to learn new languages (and programming techniques)
  - Some languages are similar; easy to walk down a family tree
- Understand implementation rationales and costs
  - Choose between alternative ways of doing things, based on knowledge of what will happen underneath
- Gain a deeper understanding of the overall concept of programming

PROGRAMMING LANGUAGE CLASSIFICATION

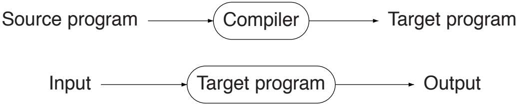
Programming languages are grouped as follows:

- Imperative
  - von Neumann → Fortran, Pascal, Basic, C
  - Object oriented → Smalltalk, Eiffel, Java, C++
  - Scripting languages → Perl, Python, JavaScript, PHP
- Declarative
  - Functional → Haskell, ML, (somehow: Scheme, Common Lisp)
  - Logic & constraint-based → Prolog, VisiCalc, RPG

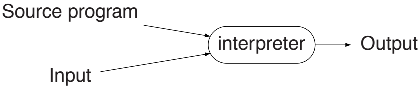
COMPILATION AND INTERPRETATION

No complier = no programmming language!

- Pure compilation: The compiler translates the high-level source program into an equivalent target program (typically in machine language), then goes away:

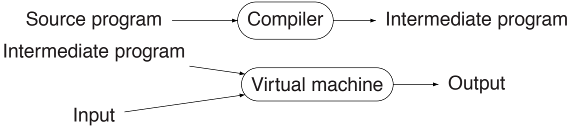


- Pure interpretation: The interpreter stays around for the execution of the program
  - The interpreter becomes the locus of control during execution



- Interpretation offers greater flexibility and better diagnostics, but compilation offers better performance

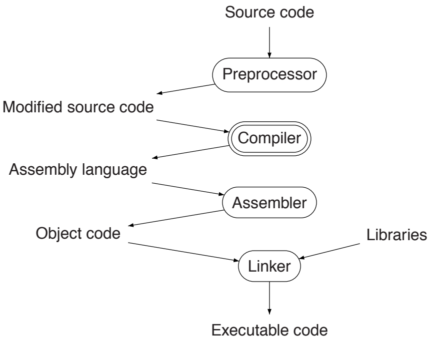
- A common case is compilation or simple pre-processing, followed by interpretation
  - Many language implementations include a mixture of compilation and interpretation



- Compilation does not have to produce machine language for some hardware
  - Compilation = translation from one language into another
  - Some compilers produce nothing but virtual instructions (Pascal P-code, Java byte code, Microsoft COM+)
- Compilation possibly preceded by a preprocessor

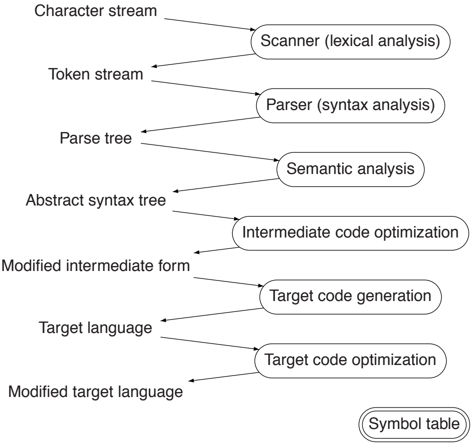
COMPILATION WORKFLOW

- For languages that compile to executable code:



- For languages that run on a virtual machine: the assembler and linker part are replaced by an interpreter (or virtual machine)

PHASES OF COMPILATION



- Scanner: divides program into "tokens" (smallest meaningful units)
  - Driven by regular expressions
- Parser: discovers the syntactic structure of a program
  - Driven by context-free grammar
- Semantic analysis: discovers the meaning of the program
  - Static analysis
  - Some other things can only be figured out at run time
- Intermediate form: tree-like structure and/or some machine-like language (but machine independent)
  - Often a form of machine language, but for an idealized machine

Phases of compilation (cont'd)

- Intermediate code optimization: produce code that does the same thing, only faster
  - Algorithmic optimization
- Code generation: produces assembly language for the target machine
- Code optimization: machine-specific optimizations (use of special instructions or addressing modes, reorder instruction to improve the load on superscallar architectures, etc.)

- Symbol table: all phases rely on a symbol table that keeps track of all the identifiers in the program and what the compiler knows about them
  - This symbol table may be retained (in some form) even after compilation has completed, for use by a debugger