General intro

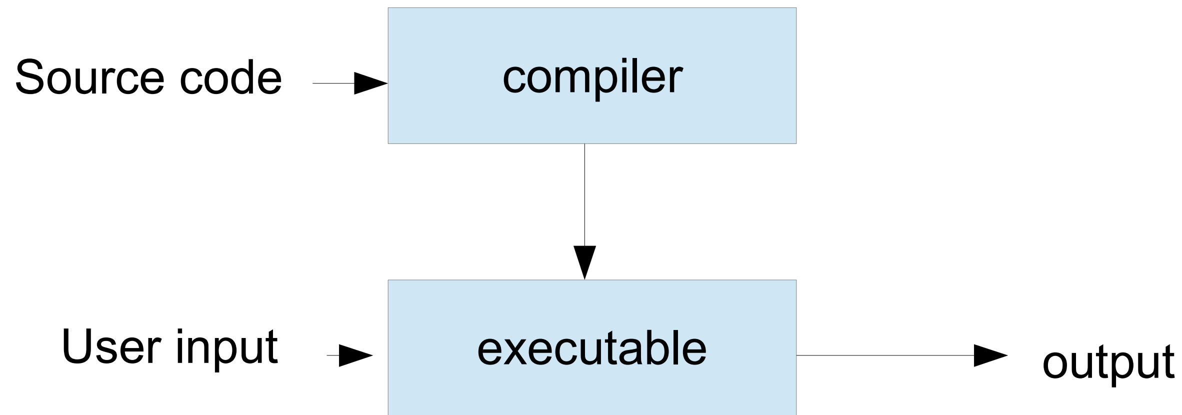
- Compiler is just another software program
- Primary goal: read a program written in one language (source language) and translate to equivalent program in another language (target language)
- Secondary goal: report any errors detected in the source
- Needs to “understand” both the source and target language
- If target language is machine code (i.e. to produce an executable) then detailed hardware architecture knowledge required

Common variations

- Compile to an executable: read entire source code program and generate machine code equivalent for target platform
- Compile to an intermediate form (e.g. Java byte code): read entire source code program and generate intermediate representation, which can later be run by a compatible virtual machine
- Act as an interpreter: read source code statements and translate them one by one

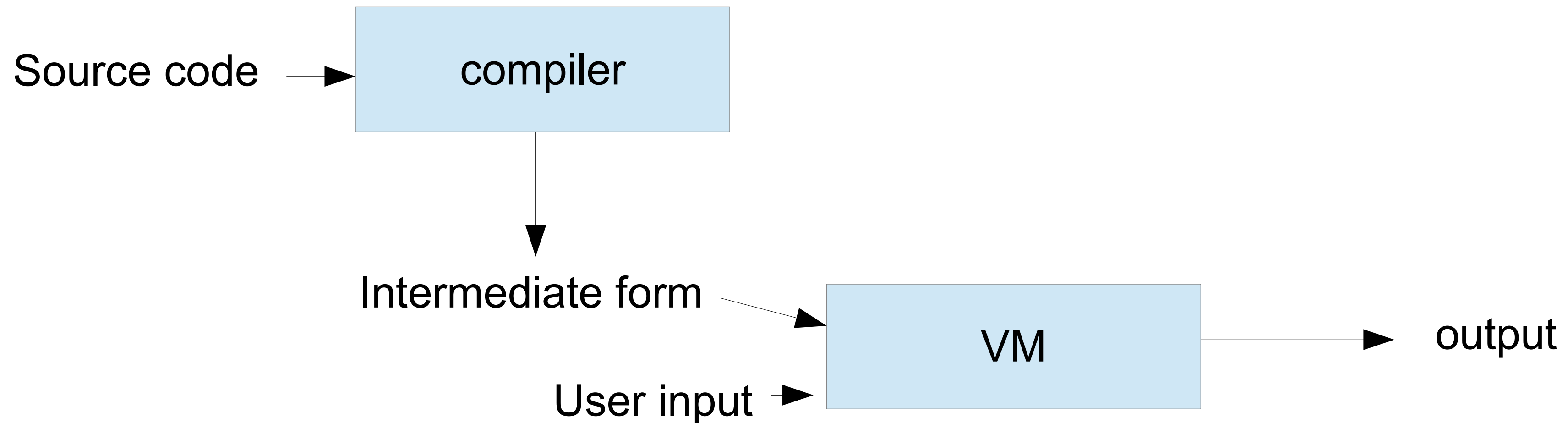
Compile to executable

- Translate to machine code executable for target platform



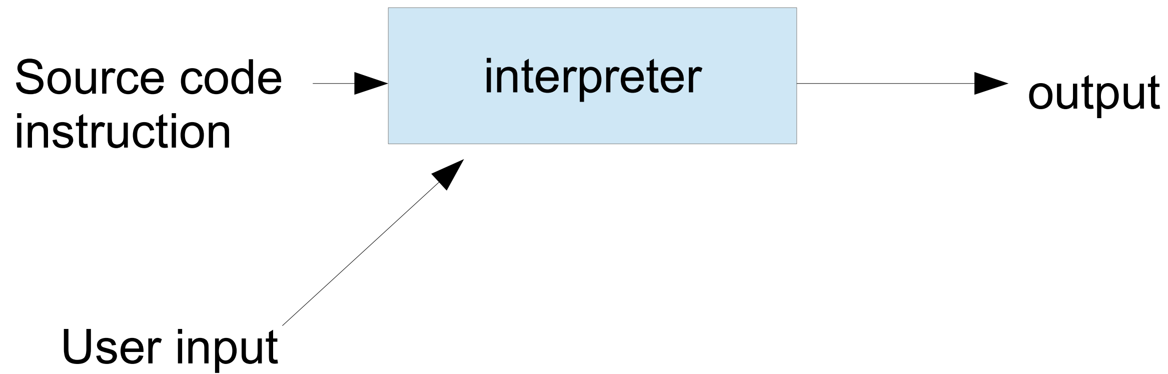
Compile to intermediate form

- Translate to intermediate representation (e.g. Byte code) to be run by a virtual machine later



Act as interpreter

- Read and execute instruction by instruction



Supporting programs

- Often incorporated into a software package along with the compiler, appears as a single unit to the user, example:
- Preprocessor: reads source code, applies macros to transform (e.g. C #defines, C++ templates, lisp macros)
- Compiler: translate to target assembly language (e.g. .cpp files to .s files)
- Assembler: translate assembly to relocatable machine code (e.g. .s files to .o files)
- Linker/loader: take set of machine code files, join together to create final executable, put in memory and finalize addresses

Common sequence of steps

- Read character sequence from source code (and preprocess?)
- Lexical analysis: produce token stream
- Syntactic analysis: produce syntax tree
- Semantic analysis: refine/augment syntax tree
- Intermediate generator: produce language independent form
- Machine-independent optimization: refine independent form
- Code generation: actual target code produced
- Machine-dependent optimization: optimize target code

Notes about the steps

- Each step is a complex process on its own with many options/variations (e.g. code optimizations encompasses a huge range of possibilities)
- Often we must maintain/share data across the steps (e.g. through the symbol table, to be discussed soon)
- The division between steps is often blurry, handled differently by one compiler than another
- Describing the processes involved can be complex, will often develop notations/abstract syntax as an aid in this

Compiler support tools

- Many tools exist to automate the construction of compilers for a language, usually based on the existence of a formal grammar for the language
- Scanner generators: read a grammar and produce a lexical analyzer (tokenizer)
- Parser generators: read a grammar and produce a syntax analyzer (parser)
- Code generation tools: read a rule collection and produce a generator that produces target code from an intermediate representation

Compiler creation tools

- Many tools exist to automate the construction of compilers for a language, usually based on the existence of a formal grammar for the language
- Scanner generators: read a grammar and produce a lexical analyzer (tokenizer)
- Parser generators: read a grammar and produce a syntax analyzer (parser)
- Code generation tools: read a rule collection and produce a generator to produce target code from an intermediate representation of a program

Compilers, languages, architecture

- Evolution of programming languages and computer architectures drive the evolution of compiler techniques (parallel processors, pipelining, memory heirarchies, etc)
- **how** we implement a language feature on a target machine has a huge impact on the feature's usefulness (e.g. early years of C++ were plagued by problems in developing compilers that could effectively meet the language specifications)
- requires strong skills/knowledge in software engineering, data structures, algorithms and complexity, hardware and architecture, language theory, dynamic programming,...