Dennis George
CS 360 Homework 1
July 6, 2020

1. What is the difference between machine language and assembly language?
   Machine language is the sequence of bits that directly controls a processor and is not human readable. Assembly language represents the operations that are used with mnemonic abbreviations so that writing programs would be less error prone.

2. In what way(s) are high-level languages an improvement on assembly language? Are there circumstances in which it still make sense to program in assembler?
   High level languages make the programmer's job of creating and maintain programs a lot easier. For some high-performance centered tasks, programming in assembly may make sense, because you have direct control for optimizations.

3. Why are there so many programming languages?
   Evolution – Computer science is constantly growing as it is a young field, and people are constantly finding better ways to do things.
   Special Purposes – Some languages have been designed to solve a problem in a specific domain.
   Personal Preference – Since programmers have a wide variety of preferences, it is highly unlikely that anyone will ever develop a universally acceptable programming language.

4. What makes a programming language successful?
   Expressive Power – One language may be considered more "powerful" than another based on its features.
   Ease of Use for the Novice – Some languages are more successful because they have a very low "learning curve"
   Ease of Implementation – Some languages are successful because they can easily be implemented on many different machines
   Standardization – Standardization is the only effective way to ensure the portability of code across platforms.
   Open Source – Most programming languages today have at least one open-source compiler or interpreter.
   Excellent Compilers – Some compilers do an unusually good job of helping the programmer manage very large projects.
   Economics, Patronage, and Inertia – Some languages remain widely used long after "better" alternatives are available, because a huge base of installed software and programmer expertise, which would cost too much to replace.

5. Name three languages in each of the following categories: von Neumann, functional, object-oriented. Name two logic languages. Name two widely used concurrent languages.
   von Neumann – C, Ada, Fortran
   functional – Lisp/Scheme, ML, Haskell
   object oriented – Smalltallk, Eiffel, Java
   logic – Prolog, SQL
   concurrent – Java, C#

6. What distinguishes declarative languages from imperative languages?
   Declarative languages hide implementation details while imperative languages embrace them.

7. What organization spearheaded the development of Ada?
   The United Stated Department of Defense.

8. What is generally considered the first high-level programming language?
   FORTRAN

9. What was the first functional language?
   LISP

10. Why aren't concurrent languages listed as a separate family in Figure 1.1?
    Most concurrent programs are currently written by adding some concurrent functionality to a sequential language, which means that there may not be a 100% concurrent language.

11. Explain the distinction between interpretation and compilation. What are the comparative advantages and disadvantages of the two approaches?
    A compiler translates a high-level source program into an equivalent target language which is typically machine code.
    An interpreter implements a virtual machine whose "machine language" is the high-level programming language. This interpreter reads statements in that language one at a time, executing them as the program goes along line by line.

12. Is Java compiled or interpreted (or both)? How do you know?
    Java could be considered as both, because a compiler produces java byte code, which is then interpreted by the java virtual machine.

13. What is the difference between a compiler and a preprocessor?
    A preprocessor simply removes comments and white space, groups characters together into tokens such as keywords, identifiers, numbers and symbols, as well ass simple analysis of syntactic structures. A compiler employs thorough analysis and nontrivial transformations.

14. What was the intermediate form employed by the original AT&T C++ compiler?
    C++ implementations based on the original AT&T compiler generated an intermediate program in C instead of assembly language.

15. What is P-code?
    P-code is a stack-based language like the byte code of modern Java compilers.

16. What is bootstrapping?
    Bootstrapping is a method in which a simple implementation of an interpreter evolves to build more complex versions until the compiler has been built.

17. What is a just-in-time compiler?
    A just-in-time compiler is a compiler that translates byte code into machine language immediately before each execution of the program.

18. Name two languages in which a program can write new pieces of itself “on the fly.”
    Lisp and Prolog.

19. Briefly describe three “unconventional” compilers—compilers whose purpose is not to prepare a high-level program for execution on a general-purpose processor.
    Two unconventional compilers are TEX and Troff – these compilers translate documents of higher level into commands for either a laser printer of typesetter using a text editor. Another unconventional compiler would be a query language processor for a database system.

20. List six kinds of tools that commonly support the work of a compiler within a larger programming environment.
    Assemblers
    Debuggers
    Preprocessors
    Linkers
    Editors
    Pretty Printers

21. Explain how an integrated development environment (IDE) differs from a collection of command-line tools.
    You can write an entire program in an IDE without having to call multiple tools with command-line tools.

22. List the principal phases of compilation, and describe the work performed by each.
    Scanner - translates characters into tokens removing white space and comments
    Parser - organizes tokens into a parse tree following a context-free grammar
    Semantic Analysis - understands parse tree to generate intermediate code and checks constraints with the help of a symbol table
    Target Code Generation - translates intermediate form into target language (machine language or assembly language)

23. List the phases that are also executed as part of interpretation.
    Scanner
    Parser
    Semantic Analysis and intermediate code generation
    Tree-walk routines.

24. Describe the form in which a program is passed from the scanner to the parser; from the parser to the semantic analyzer; from the semantic analyzer to the intermediate code generator.
    Token stream, parse tree, abstract syntax tree or intermediate form.

25. What distinguishes the front end of a compiler from the back end?
    The front end may be shared by compilers for more that one language (target language), and the back end may be shared by compilers for more than one source language.

26. What is the difference between a phase and a pass of compilation? Under what circumstances does it make sense for a compiler to have multiple passes?

Each phase discovers information of use to later phases or transforms the program into a form that is more useful to the subsequent phase.

A pass is a phase or set of phases that is serialized with respect to the rest of compilation: it does not start until previous phases have completed, and it finishes before any subsequent phases start.

27. What is the purpose of the compiler's symbol table?

It is a data structure that maps each identifier to the information known about it including the identifier's type, internal structure, and scope.

28. What is the difference between static and dynamic semantics?

Of course, not all semantic rules can be checked at compile time, these are static semantics Those that must be checked at run time are referred to as the dynamic semantics of the language.

29. On modern machines, do assembly language programmers still tend to write better code than a good compiler can? Why or why not?

For modern microprocessor architectures, particularly those with so-called superscalar implementations (ones in which separate functional units can execute instructions simultaneously), compilers can usually generate better code than can human assembly language programmers.