Chapter 1 - Introduction

September 18, 2017

Computing without digital computers

- Analog machines
  - Antikythera device
  - Pascal's calculators
  - Babbage's difference engine and Babbage's analytical engine
  - The Lehmers' factoring machines,
  - Various military machines, such as the famous Norden bombsight
- Hybrid systems
- Quantum computing

Turning to Turing-class machines

- (See handout)

Our Digital World: Stored Program Computing

- With digital computers using the stored program paradigm, it's all bits: data and instructions are *not* different things.
- General purpose computers of this ilk have the ability to treat instructions as data.
- Comparable to a Turing Machine

Machine language is just the bits:

- Example from /usr/bin/emacs ("xxd -b")

```
0000000: 01111111 01000101 01001100
         01000110 00000010 00000001  .ELF..
0000006: 00000001 00000000 00000000
         00000000 00000000 00000000  ......
000000c: 00000000 00000000 00000000
         00000000 00000010 00000000  ......
```

Improving to Assembly language

- We can give more human-language-like names to instructions
- We can give logical names to memory locations

Improving to Assembly language

- More advanced assemblers allow provide powerful facilities to do computations on these logically named memory locations
- More advanced assemblers allow macros in-house (of course, you can also use external macro rewriters like m4) and allow for larger sets of directives (think of them as highly useful pragmas)

Improving to Assembly language

- Example from handwritten assembly assembled and then disassembled ("radare2")

```
0x004000b0    4d31d2          xor r10, r10
0x004000b3    488b5c2408      mov rbx, [rsp+0x8]
0x004000b8    488b542408      mov rdx, [rsp+0x8]
0x004000bd    8a0c250f076.    mov cl, [0x60070f]
0x004000c4    443813          cmp [rbx], r10b
0x004000c7    7417            jz 0x4000e0
```

Advantages of assembly language

- Total freedom, full power of the hardware, including truly self-modifying code

Advantages of assembly language

- Knuth's arguments for MIX/MMIX (TAOCP, 3rd edition, p. ix):
  - Programmers' tendency to write to a language's idioms: "A programmer is greatly influenced by the language in which programs are written; there is an overwhelming tendency to prefer constructions that are simplest in that language, rather than those that are best for the machine."
  - High-level languages are inadequate for discussing low-level concerns.

Advantages of assembly language

- Knuth's arguments for MIX/MMIX continued
  - People should understand the machines that they use.
  - New languages come into and go out of fashion quite rapidly, but notions of state manipulation expressed as an idealized machine language don't.
- See also MMIX

Advantages of assembly language

- Can avoid glibc! ;-)

Limits of assembly language

- Harder to maintain (look at how long it is taking to migrate from MIX to MMIX!)
- Processor instructions go through generations, both inside and outside processor families
- Inherently imperative coding style

The Analogy of Communication between People and between People and Computers

- Human languages are one of the most importants of communication we possess
- Programming languages share many of the same characteristics that we desire from human languages
- There are lots of human languages, and lots of computer languages

The Art of Language Design

- Why we change things: we learn better ways and evolve
- Specialization in a particular problem space:
  - Look at TeX/LaTeX
  - Graphviz's various dialects
  - Perl's regular expressions
  - Emacs elisp
  - All look at a particular problem space

The Art of Language Design

- Personal preference
  - For instance, some people find Python's indenting rules obnoxious
  - Other people are horrified by cryptic languages like APL or by Perl's tersest modes

What helps makes for success in a programming language

- Expressiveness!
  - Ability to abstract at the most useful level for the problem space
  - For instance, these slides are in an ultra-simple “markdown” language which seems reasonably appropriate

What helps makes for success in a programming language

- Easy to learn
  - Some languages are just plain easy to learn: Basic, Pascal, Logo, Scratch can all be picked up even by the earliest students; advanced students can absorb one of these languages in a very short time

What helps makes for success in a programming language

- Ease of re-implementation: if it can be ported quickly, it has big advantages (example BLISS (very hard to impossible to port) versus Pascal (very easy to port via pcode).
- Standardization: widely accepted languages generally have official standards and/or canonical implementations.
- Open source

What helps makes for success in a programming language

- Excellent compilers / interpreters
  - Fortran family
- Economics, patronage, inertia, (and luck!)
  - IBM and PL/I
  - DOD and Ada
  - Microsoft and C#

The Programming Language Spectrum

- Declarative languages: these languages aspire to "Do What I Want"; the classic example is the logic programming language Prolog (1970).

The Programming Language Spectrum

- Declarative languages, continued
  - Functional languages: while APL is an early ancestor, the functional languages had their first strong impetus from the publication of Backus's Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. Classic examples here are languages like Standard ML and Haskell. These languages shy away from the traditional variable assignment paradigm.

The Programming Language Spectrum

- Declarative languages, continued
  - Dataflow languages: So far, these type of languages have not proven to be popular. However, they do have some interesting and attractive properties.
  - Logic programming languages: the classic example, as mentioned previously, is Prolog. However, there are more modern languages, such as the hybrid functional logic programming languages Curry and Mercury.

The Programming Language Spectrum

- Imperative languages: these are "Do What I Say" (maybe aspiring to "Do What I Mean", as, for instance, Perl does) languages.
  - "von Neumann": the highly successful and well-established family that includes C, Ada, Fortran, and all those other languages that are based on variable assignment. These languages are conceptually all quite close to the actual hardware being used.

The Programming Language Spectrum

- Imperative languages, continued
  - Scripting languages: Perl, Python, Javascript, Ruby, PHP, and their ilk. These languages usually have outstanding library capabilities; they are usually reasonably easy to learn, and usually are the most suitable languages for one-off, lightweight programming tasks.
  - "Dictionary" languages: the classic here is Forth, which is an "untyped" language based on defining words that (largely) manipulate stacks.

The Programming Language Spectrum

- Imperative languages, continued
  - Object-oriented languages: Smalltalk is the classic example here, a very pure object-oriented language that inspired Objective C, C++, Java, and a host of other languages.

Why study programming languages?

- It's fun! It's interesting! It's even practical.
- The obvious answer is that it is easier to make informed decisions about what language to use for what purpose. "When you only have a hammer, everything looks like a nail" is very true in computer science; learning to use a full suite of appropriate languages can make your life as computer scientist more fulfilling.

Why study programming languages?

- Just like learning human languages, learning more programming languages makes it easier to pick up new ones. (Though sometimes the paradigm shifts (even the syntax shifts) can be a bit daunting.)

Why study programming languages?

- Make better use of the lower level bits.
  - In system administration, I constantly emphasize "Understand the system calls, these are key to understanding the operating system's activities and program interactions."
  - The same is true with programming languages, particularly when you want better performance; understanding garbage collection issues, for instance, can make a very large difference in eaking out better performance.

Why study programming languages

- Simulate useful features in languages that lack them. Older languages can benefit from newer concepts and techniques, and sometimes to the easiest way to that benefit is to just simulate the feature.
- Techniques developed for parsing programming languages can also be used for other arenas, such as parsing configuration files. I have written more Bison parsers for configuration files than I ever have for my programming language experiments.

Compilation and interpretation

- More of a spectrum than a Manichaean duality.
- In the large, compilers *transform* your source code into a new form directly executable by a processor, generally called something like “binary form”, or “executable form.”
- In the large, interpreters *execute* your source code. There are a variety of ways of doing this, some coming quite close to compilation.

Compilation and interpretation

- REPLs (read-evaluate-print loop) have become a popular adjunct to the idea of interpretation, blending the roles of programmer and user into a nice mix of interactivity. These are especially condign with languages in the ML family, as you will see in some of the exercises.

Compilation and interpretation

- Implementations generally are considered as “compilers” if the translator does a lot of work, and “interpreters” if the translator is less taxed.
- An amusing example from page 19 of the text: some Basic interpreters actually recommended removing comments to improve performance. Remember, a Basic interpreter has to re-read and then discard those comments each and every time it runs, and lots of comments are lots of unused bytes that must be dealt with each execution of the code.

Compilation and interpretation

- The last paragraph of page 20 in the text is not clearly worded. The potential “changes in the format of machine language files” is evidently referring to changes in binary formats such as ELF.

Compilation and interpretation

- The initial compilation phase for C is a preprocessing one, with macros much like an advanced assembler like yasm would provide. and conditionals, which let you literally remove inapplicable code (ifdef and its variants.) You can stop the gcc C compiler at this phase with "gcc -E"
- The next phase in C compilation is to turn out assembly language code ("gcc -S"); the final phases don't involve the compiler: 1) have the assembler turn this into machine language and 2) have the linker turn all of this into a final binary format for direct execution.

Compilation and interpretation

- Some natively interpretive languages also allow for compilation; these languages often have late binding properties that will need the equivalent of interpretation even for "compiled" code.
- Your text on page 23 labels TeX as a compiler, which is largely true, but pure TeX only goes as far as a file format called DVI, which is not a format understood by any modern printer. DVI can be converted to PDF or PostScript by programs such as dvips and dvipdf. (It is true that there are versions of TeX such as pdftex/pdflatex which can turn out PDF directly.)

Overall view of compilers

- Compilation is one of the great practical triumphs of computer science.
- Typical phases of compilation:

1. Scanning (lexical analysis)
2. Parsing (syntax analysis)
3. Semantic analysis and intermediate code generation
4. Maybe some optimization
5. Target code generation
6. Maybe some code improvement on the final code (such as peephole optimization)

Lexical and syntax analysis

- Lexical analysis (scanning) is a generally straightforward task; while there are tools to generate "lexers" such as flex or re2c, lexers are often written by hand. All a lexer typically needs to do is remove comments and identify tokens.

Lexical and syntax analysis

- Syntax analysis (parsing) takes the tokens generated by the lexer and organizes them into a parse tree that represents the constructs of the language being recognized. These constructs are expressed as a context-free grammar; generally, such parsers are either written as a recursive descent parser (generally by hand, though automation tools do exist), or a bottom-up parser (generally automated by tools like bison or lemon, or perhaps along the lines of antlr.)

Semantic analysis

- Generally a parser will build a parse tree; during this build, various semantic actions can be triggered by semantic action rules embedded in the parser. Other semantics can be done later, perhaps by rewalking the tree and massaging it into more refined (and perhaps more optimized) forms, or later, as dynamic semantic routines invoked at execution.
- Target code generation: a compiler can generate assembly code (and many do), or it might generate an intermediate form like LLVM intermediate representation.