Compiler Design

William A. Barrett
San Jose State University
CmpE 152

FALL 2005 VERSION

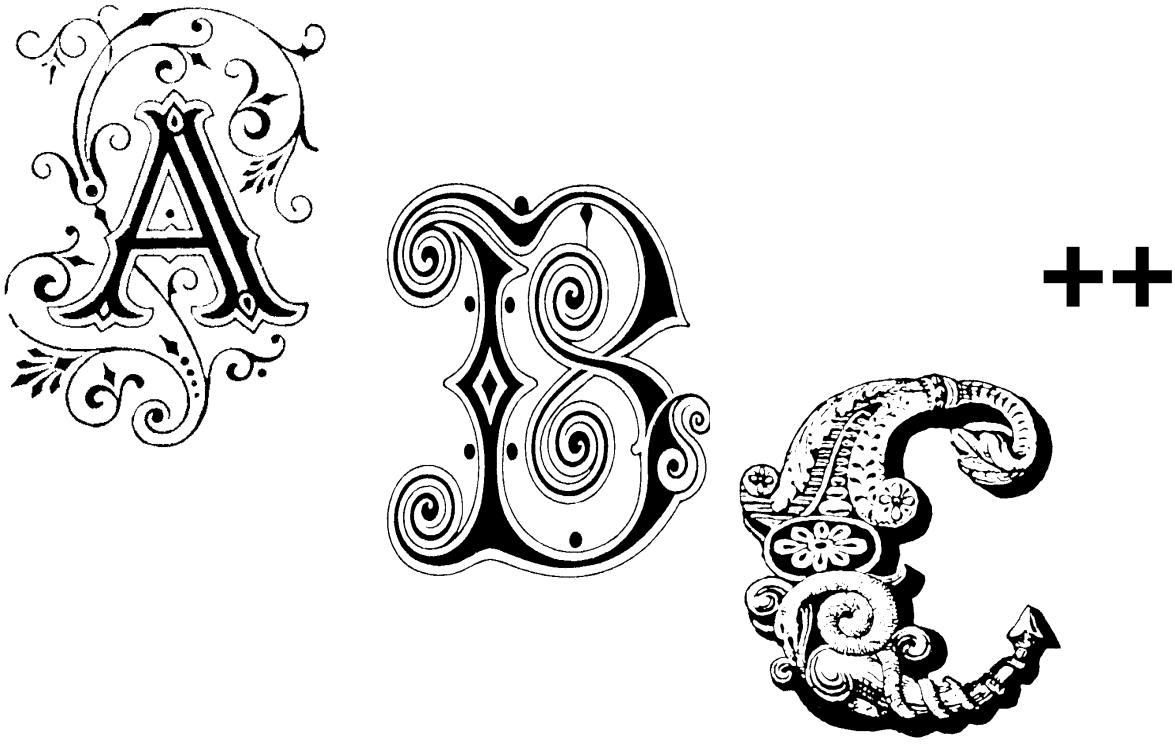

Table of Contents

Chapter 1: Writing Compilers and Interpreters

William A. Barrett, San Jose State University

nch1.doc

What's a Compiler?

Most programmers write software in a high-level language, such as Pascal, C, C++, java, Cobol and others. Many are not aware of the fact that they are really making use of a sophisticated program called a *compiler* that bridges the gap between their chosen language and a computer architecture. Some have no concept at all, or a very poor grasp of the computer's instruction set, memory organization, and other details that make their software work. A compiler therefore provides a valuable form of *information hiding.* Most programmers don't need to know anything about the details of translation and execution, only the properties claimed to be supported by the high-level language and its libraries.

A professional computer engineer cannot afford to be ignorant of the details. Compilers are a valuable and reliable tool for the most part, but no software is perfect. If a bug arises in code, it's sometimes useful to be able to trace it down into the assembler/micro instruction level. High performance may require certain low-level operations. Finally, knowing something about the strategy used by a compiler helps the engineer understand the high-level language at a deeper level.

We use the word *translation* in roughly the same sense as a person learns to translate from English to (say) German. Knowing the German equivalent of lots of English words is an important first step, and this may be the most difficult step for a person to take. However, there's more. Good German requires paying attention to gender in all nouns. A *chair*, a *pencil*, or an *automobile* in the Romance languages is either masculine, feminine, or neutral. The gender determines just how adjectives and verbs may be applied to the nouns. Also, the ordering in a German sentence is different than English. Germans love to place the verb at the end of the sentence, whereas in English, it's between the subject and the predicate.

Throw father the stairs his coat down might be the way a German would express the sentence *Throw father's coat [to him] down the stairs*.

The alphabets are slightly different, though both English and German conform to the Latin alphabet to a considerable degree. Spoken German is yet another problem, since there are certain voice inflections that are unfamiliar and uncomfortable for the English speaker; also the words take on a pronunciation unlike that of English: letter w becomes a v sound, for example, so that *wolf* is spoken as *vuulf*.

All in all, it's obvious to anyone who's studied a foreign language that you can't become fluent (or even well understood) in the language by just changing all your English words into the foreign equivalents. This is also why machine translations of natural language are at worst just wrong and at best, seem stilted and unnatural.

A similar problem occurs in understanding the function of a compiler. Just changing names and operators into some kind of machine language equivalents isn't sufficient. Some of the features in the high-level language, such as types, don't even show up in the translation per se. They just influence the way in which other features are translated. We also must face up to understanding the organization of the target machine, its instruction set, and also know everything about the conventions of the symbolic assembler that the compiler intends to generate.

In fact, we can argue that in order to design and implement a useful compiler, one must be reasonably expert in *all* these disciplines:

- the *source language* (what language we're trying to translate, i.e. Pascal),
- the *host language* (what programming language we're using to write the translator, i.e. C++),
- the *host metalanguages* (special little languages that describe tokens or language structure),
- the *target language* (what we're translating into, i.e. Microsoft symbolic assembler), and
- the *target machine* (what CPU will finally execute our program).
- compiler *tools,* the algorithms and data structures that are commonly used to design a compiler.

If we have a poor understanding of any one of these, our effort at writing a compiler is doomed.

Why High-level Languages?

Let's review some of the more important reasons for using a high-language with a compiler, compared to directly coding assembler for a processor.

Note first that an assembler such as MASM is itself a fairly sophisticated translator. Although in general an assembler converts one line of source code into one instruction, there are a large number of specialized operations required of a modern assembler.

However, the most important case that can be made for a high-level language is that

It bridges the gap between machine instructions and humans

Regarding machine instructions and assembler:

- Instructions are primitive. Very simple operations, usually one line per instruction.
- Coding requires close attention to status bits and machine environment.
- Coding requires working with a finite set of registers and memory rather than mathematic entities.
- Addressing is very complicated and hard to get correct.
- Need hundreds of instructions to carry out a conceptually simple operation
- Hand-written assembly code locks you into a particular machine and machine version. Very hard to change assembler to a newer or more powerful machine
- ...but assembler is essential for certain low-level operations and/or highest performance.

A high-level language and its compiler:

- Provides much higher productivity in software development.
- Approaches the performance of assembler for most operations.
- Provides an easy way to express complex operations.
- Provides modular decomposition of very complex problems into simpler components

- Provides ease of checking correctness by structured development and inspection
- Provides warnings of syntax and typing errors.
- Guarantees correctness of underlying assembler, if the source code is correct.
- Provides portability. All or most of a program can be recompiled on different computers.
- Can provide inline assembly for critical regions. (but assembler defeats portability!)

Everything Goes Through a Compiler

Almost every piece of software used on any computer has been produced by a compiler.

This astonishing claim arises from the extreme difficulties faced by any human in writing absolute binary instructions for any computer. The "compiler" used may be rather primitive, perhaps just a simple symbolic assembler that maps symbols directly to instructions. But the use of symbols to specify computer operations is nearly universal. Just consider:

- Nearly every component of a typical operating system--its kernel, libraries, support utilities, etc.--are typically written in a high-level symbolic form, and translated through a compiler to machine form.
- Database management systems are written in high-level language form. The popular SQL database access language is an interpreted language: some machine translator is needed to make sense of its directions, and turn them into useful actions.
- HTML is an interpreted language--the "L" stands for "language": *HyperText Markup Language.* It's a powerful way to display text, images, forms, menus, and more. And it's fairly easy to write directly with a text editor.
- The popular HTML *browsers*, for example those written by Netscape and Microsoft, were written in a high-level language, most likely C++ or java. By using a high-level language, it was possible to almost automatically port the browser to a large number of different machine platforms with little additional human effort.
- Compilers themselves are written in a high-level language, sometimes in the source language of the compiler itself! A compiler for C was organized in a modular fashion years ago by Steve Johnson to make it easier to generate code for a variety of different machines. The popularity of C as a language largely stems from this effort. Once a reliable C compiler is ready for the platform, the Unix operating system and large number of libraries can be generated for the new platform, using that compiler as a basis.
- New microprocessor designs require one or more compilers for the machine. These compilers usually start with a *cross-compiler,* which runs on one established platform, but generates machine code for the new system. The cross-compiler is written in a high-level language using an established compiler. Again, C is popular for this kind of effort.
- The Gnu C/C++ compiler was designed to service almost any hardware platform. In general, it relies on a C compiler and a few libraries already provided by the machine's manufacturer, and uses these to construct a Gnu compiler. We should point out that each manufacturer's C compiler and library is different in subtle ways. The Gnu team therefore wrote an extensive set of little tests to see just how a particular platform differs from a standard one, then used that configuration information to develop a specialized (and optimized) compiler for the platform. With that compiler, many other Gnu tools written in its standard C language can be compiled for the platform.

- The Linux operating system rests on the ingenuity of the Gnu team in developing a standard platform based on their C compiler. A version of Linux is now available on a wide variety of platforms, including Pentiums, Sparc, most RISC machines and others. About the only requirement is that the platform processor support 32-bit addressing and some key operating system features, such as multitasking, hardware protection, user vs. system modes.

Productivity

Software development is an economic activity (someone has to pay for it and someone expects to get paid for their skilled time). Productivity with a suitable high-level language and its compiler tools is easily 50 to 500 times the productivity in assembler.

The software development cost of a typical modern product may exceed the hardware development cost by a large factor, and can be very expensive for even a modest technical product. There's also a strong demand to get the software written and tested within an aggressively short deadline, because otherwise the competition may release a similar product first and capture the bulk of the market.

Many large software products are expected to operate on different platforms, including some that haven't been developed yet. Having the code portable through being written in a high-level language is a major factor in reducing the cost or porting software to new platforms.

All these considerations clearly call for the most productive approach to software development possible, with the software written in high-level languages that provide for the maximum in safety (freedom from bugs), portability, yet without compromising performance.

Why Study Compiler Theory?

We're going to explore just how a compiler is designed and organized. We'll see that its key components involve a kind of expert system, using *production rules.* The central challenge in a compiler is *making sense of a long sequence of characters in a source file.* This a challenging problem in computing, and in fact, is a form of artificial intelligence: how to map something that's clear and sensible to a *human* into something that's clear and sensible to a *machine.*

Many people have dreamed of an intelligent machine such as HAL, the spaceship robot in the movie *2001-A Space Odyssey.* HAL could talk to the human crew members, could understand their verbal commands, could provide intelligent criticism of their attempts at artwork, could compose and criticize artwork, and could even read lips! Very few of those human-like capabilities have actually been achieved in a machine, and of those that have, the machine falls far short of humans in performance. For example, you can often respond to a machine over the telephone through a voice message: "Say or press one". That's a form of speech recognition, and the machine recognition can figure out one out of a small number of responses. But it's useless to ask that machine to connect you to someone in the auditing department by name!

The intelligence involved in a compiler is far more modest than reading lips or listening to speech. We meet the machine halfway by providing it with a file of ASCII characters. We also expect that the machine may not make sense of just anything written in that file; only certain carefully organized sequences of characters will do. Anything else is considered a "syntax error". As a human, it's up to us to figure out why the machine recognition failed, and then repair the error. Computers still *do what they are told, not what we want.* Yet even that limited form of AI is very useful to us. With a limited amount of study on our part, *we* can tell a machine what we want it do.

The value to a computer engineering of understanding some compiler theory lies in these general observations:

- Virtually every transaction humans make with a machine is through some form of translation,