Compilers Overview

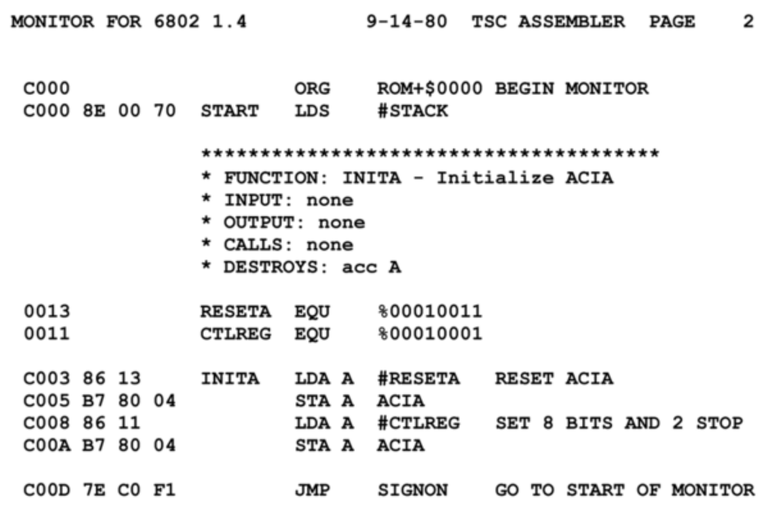
Christopher League*

20 January 2016

What is a compiler?

Early machines were programmed painstakingly in bits, entered using octal or hexadecimal. For example, look at the PDP-8 or PDP-11 console: switches grouped into threes that made it easy to enter octal numbers directly into memory. (Eventually these machines could also connect to tape drives and teletypes for more convenient data and code entry.)

Entering numbers directly is inconvenient and error-prone, so we started creating *assemblers.* An assembler is a program to translate text *op-codes* into binary. For example, here is a document showing hexadecimal numbers (such as `B7 80 04`) next to corresponding text op-codes (`STA A ACIA`).



```
MONITOR FOR 6802 1.4          9-14-80  TSC ASSEMBLER  PAGE    2

 C000                  ORG    ROM+$0000 BEGIN MONITOR
 C000 8E 00 70   START LDS    #STACK

                 ****************************************
                 * FUNCTION: INITA - Initialize ACIA
                 * INPUT: none
                 * OUTPUT: none
                 * CALLS: none
                 * DESTROYS: acc A

 0013            RESETA EQU   %00010011
 0011            CTLREG EQU   %00010001

 C003 86 13      INITA  LDA A #RESETA   RESET ACIA
 C005 B7 80 04          STA A ACIA
 C008 86 11             LDA A #CTLREG   SET 8 BITS AND 2 STOP
 C00A B7 80 04          STA A ACIA

 C00D 7E C0 F1          JMP   SIGNON    GO TO START OF MONITOR
```

Figure 1: Hexadecimal encoding of Motorola 6802 assembly language

The assembly language is especially convenient because it can figure out relative and absolute offsets for jumps and data using text labels (like `INITA` and `CTLREG`). However, the statements themselves are just a 1:1 mapping of machine instructions – no higher-level abstractions.

So the audacious idea of the compiler is to allow programmers to write higher-level code – closer to the problem domain, and more natural notation – and have a program translate it automatically into assembly or machine code.

*Copyright 2016, some rights reserved (CC by-sa)

One of the absolute pioneers in compilers is Grace Hopper, who designed the languages MATH-MATIC (a predecessor of FORTRAN) and FLOW-MATIC (a predecessor of COBOL). She actually coined the term 'compiler' for this type of software. You absolutely should read the introduction and text of her keynote address to the History of Programming Languages conference in 1978.

- Introduction to Grace Hopper's keynote by Jean Sammet
- Keynote address by Grace Hopper

(These links are to the ACM Digital Library; you can download them when on campus or if you are an ACM member.)

Hopper fought a lot of entrenched attitudes about how computers 'should' be programmed: "In the early years of programming languages, the most frequent phrase we heard was that the only way to program a computer was in octal. Of course a few years later a few people admitted that maybe you could use assembly language."

Compiler structure

Most compilers today are multi-phase, multi-pass systems. By *multi-phase,* we mean that the source goes through several layers of analyses and transformations before emitting the target or object code. The results of those phases are themselves languages, usually referred to as *intermediate languages* or *intermediate representations* (IR).

The compiler errors you see generally come from three (sometimes four) distinct phases, although they are not always identified that way.

- A lexical error refers to something wrong with splitting your program into tokens. For example, if you forget a closing quote " character on a string, or you use an entirely invalid character (some languages only support the ASCII character set, for example).
- A syntax error refers to something wrong with the tree structure of your program. These are very common errors, like missing semi-colons, mismatched parentheses or braces, extra commas, etc.
- A semantic error (or type error) happens when you have undeclared identifiers or the types of operands don't match, such as trying to add an integer and a string.

The next three phases don't typically output error messages unless there's a bug in the compiler. There is a fourth source of errors, but it's part of the *linker,* which is logically a separate tool (depending on the language and platform). The linker takes multiple chunks of object code and merges them together into a single executable. If there are conflicting definitions or missing definitions, those show up as linker errors.

Figure 2: Grace Hopper

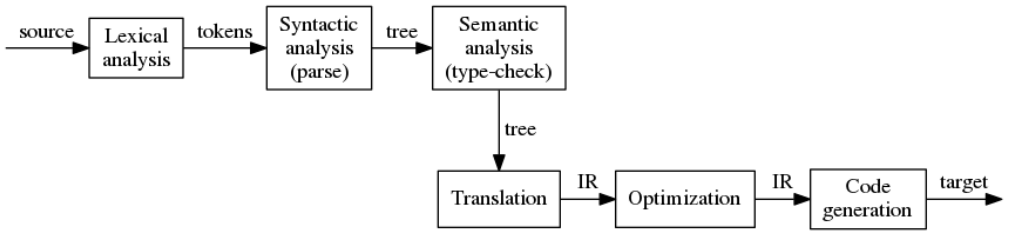


Figure 3: Compiler phases

I also said that most compilers today are multi-pass. That means they load source language file into memory and can go back and forth within the file. Early compilers, back when memories were small, were designed to type-check, translate, and generate code by just making one pass through the file. This is why some older languages like C require *forward declarations* before you call a function but newer languages like Java or Haskell can declare and call functions in any order.

```
int square(int x);   // Forward declaration (prototype)

void main() {
    printf("%d\n", square(90));
}

int square(int x) {  // Function definition
    return x*x;
}
```

Compilers don't necessarily output object (machine) code. Instead, the target language could just be another programming language. For example, the first versions of C++ were implemented as a translator to plain C code. Also there are plenty of new languages whose compilers target Javascript so they can run in web browsers.