Compiler Construction/Introduction

Introducing Compilers and Interpreters

A **compiler** is a computer program that implements a programming language specification to "translate" programs, usually as a set of files which constitute the *source code* written in *source language*, into their equivalent machine readable instructions **(the target language, often having a binary form known as object code)**. This translation process is called *compilation*. We *compile* the source program to create the *compiled program*. The compiled program can then be run (or executed) to do what was specified in the original source program.

The **source language** is always a higher-level language in comparison to machine code, written using some mixture of English words and mathematical notation, assembly language being the lowest compilable language (an *assembler* being a special case of a compiler that translates *assembly language* into machine code). Higher-level languages are the most complex to support in a compiler/interpreter, not only because they increase the level of abstraction between the source code and the resulting machine code, but because increased complexity is required to formalize those abstract structures.

The **target language** is normally a low-level language such as assembly, written with somewhat cryptic abbreviations for machine instructions, in these cases it will also run an assembler to generate the final machine code. But some compilers can directly generate machine code for some actual or virtual computer e.g. byte-code for the Java Virtual Machine.

Another common approach to the resulting compilation effort is to target a *virtual machine*. That will do just-in-time compilation and byte-code interpretation and blur the traditional categorizations of compilers and interpreters.

For example, C and C++ will generally be compiled for a target `architecture'. The draw-back is that because there are many types of processor there will need to be as many distinct compilations. In contrast Java will target a Java Virtual Machine, which is an independent layer above the 'architecture'. The difference is that the generated byte-code, not true machine code, brings the possibility of portability, but will need a Java Virtual Machine (the byte-code interpreter) for each platform. The extra overhead of this byte-code interpreter means slower execution speed.

An **interpreter** is a computer program which executes the translation of the source program at run-time. It will not generate independent executable programs nor object libraries ready to be included in other programs.

A program which does a lot of calculation or internal data manipulation will generally run faster in compiled form than when interpreted. But a program which does a lot of input/output and very little calculation or data manipulation may well run at about the same speed in either case.

Being themselves computer programs, both compilers and interpreters must be written in some *implementation language*. Up until the early 1970's, most compilers were written in assembly language for some particular type of computer. The advent of C and Pascal compilers, each written in their own source language, led to the more general use of high-level languages for writing compilers. Today, operating systems will provide at least a free C compiler to the user and some will even include it as part of the OS distribution.

Compiler construction is normally considered as an advanced rather than a novice programming task, mainly due to the quantity of code needed (and the difficulties of grokking this amount of code) rather than the difficulty of any particular coding constructs. To this most books about compilers have some blame. The large gap between production compilers and educational exercises promotes this defeatist view.

For the first version of a compiler written in its own source language you have a bootstrapping problem. Once you get a simple version working, you can then use it to improve itself.

The compilation process

At the highest level, compilation is broken into a number of parts:

1. Lexical analysis (tokenizing)
2. Syntax analysis (parsing)
3. Type checking
4. Code generation

Requirements

Any compiler has some essential requirements, which are perhaps more stringent than for most programs:

- Any valid program must be translated correctly, i.e. no incorrect translation is allowed.
- Any invalid program must be rejected and not translated.

There will inevitably be some valid programs which can't be translated due to their size or complexity in relation to the hardware available, for example problems due to memory size. The compiler may also have some fixed-size tables which place limits on what can be compiled (some language definitions place explicit lower bounds on the sizes of certain tables, to ensure that programs of reasonable size/complexity can be compiled).

There are also some desirable requirements, some of which may be mutually exclusive:

- Errors should be reported in terms of the source language or program.
- The position at which an error was detected should be indicated; if the actual error probably occurred somewhat earlier then some indication of possible cause(s) should also be provided.
- Compilation should be fast.
- The translated program should be fast.
- The translated program should be small.
- If the source language has some national or international standard:
  - Ideally the entire standard should be implemented.
  - Any restrictions or limits should be well and clearly documented.
  - If extensions to the standard have been implemented:
    - These extensions should be documented as such.
    - There should be some way of turning these extensions off.

There are also some possibly controversial requirements to consider (see chapter on dealing with errors):

- Errors detected when the translated program is running should still be reported in relation to the original source program e.g. line number.
- Errors detected when the translated program is running should include division by 0, running out of memory, use of an array subscript/index which is too big or too small, attempted use of an undefined variable, incorrect use of pointers, etc.

Overall Structure

For ease of exposition we will divide the compiler into a front end and a back end. These need not even be written in the same implementation language, providing they can communicate effectively via some intermediate representation.

The following list itemizes the tasks carried out by the front end and the back end. Note that the tasks are not carried out in any particular order, as outlined below, and discussed in more detail in subsequent chapters.

- **Front end**
  - lexical analysis - convert characters to tokens
  - syntax analysis - check for valid sequence of tokens
  - semantic analysis - check for meaning
  - some global/high-level optimization
- **Back end**
  - some local optimization
  - register allocation
  - peep-hole optimization
  - code generation
  - instruction scheduling

Almost all the source-language aspects are handled by the front end. It is possible to have different front ends for different high-level languages, and a common back end which does most of the optimization.

Almost all the machine-dependent aspects are handled by the back end. It is possible to have different back ends for different computers so that the compiler can produce code for different computers.

The front end is normally controlled by the syntax analysis processing. As necessary, the syntax analysis code will call a routine which performs some lexical analysis and returns the next token. At selected points during syntax analysis, appropriate semantic routines are called which perform any relevant semantic checks and/or add information to the internal representation.

Next - Describing a Programming Language

Retrieved from "https://en.wikibooks.org/w/index.php?title=Compiler_Construction/Introduction&oldid=3357979"

This page was last edited on 5 January 2018, at 20:05.

Text is available under the Creative Commons Attribution-ShareAlike License.; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy.