OPERATING AND PROGRAMMING SYSTEMS SERIES

PETER J. DENNING, *Editor*

A Laboratory Manual for Compiler and Operating System Implementation

Maurice H. Halstead

IE COMPUTER SCIENCE LIBRARY

A Laboratory Manual for Compiler and Operating System Implementation

Maurice H. Halstead
Computer Science Department, Purdue University,

NORTH HOLLAND·NEW YORK
NEW YORK • OXFORD

Elsevier North Holland, Inc.
52 Vanderbilt Avenue, New York, N.Y. 10017

Distributors outside the United States and Canada:

Thomond Books
(A Division of Elsevier North Holland Scientific Publishers, Ltd.)
P.O. Box 85
Limerick, Ireland

International Standard Book Number 0-444-00142-5
Library of Congress Card Number 73-10897

Third Printing, 1978

Library of Congress Cataloging in Publication Data

Halstead, Maurice Howard, 1918-
A laboratory manual for compiler and operating system implementation.

(Operating and programming systems series)
Includes bibliographies.
1. Compiling (Electronic computers) I. Title.
QA76.6.H32 001.6'425 73-10897
ISBN 0-444-00142-5

Manufactured in the United States of America

A Laboratory Manual for Compiler and Operating System Implementation

THE COMPUTER SCIENCE LIBRARY

Operating and Programming Systems Series
Peter J. Denning, *Editor*

Halstead	A Laboratory Manual for Compiler and Operating System Implementation
Spirn	Program Behavior: Models and Measurements
Halstead	Elements of Software Science
Franta	The Process View of Simulation

To Sylvia

Prologue

In studying or teaching fundamental concepts involved in both compilers and operating systems, it may be advantageous to have available a compact example of each, to serve as a source of laboratory experiments or semester projects to supplement material covered by the range of available texts in the field.

The first part of this manual provides the material required by a student to enable him to write a specific, minimal self-compiler; to check it out; to extend the source language which it accepts; and to improve the object code which it produces. The second part provides comparable material for the implementation of a skeletal time-slicing operating system. The student will also need, in addition to talent and industry, access to any computer larger than a desk calculator and other than a Univac 1108. The latter restriction arises because the examples of both the compiler and operating system given are instances of implementations for that computer, hence little would be learned by reinstalling them on that particular machine.

The objectives are threefold: (1) to give the student personal familiarity with the more detailed and specific problems which must be solved for any compiler or for any operating system, including that class of problems which appear too trivial to warrant attention in a scientific paper, or even a text, but which nevertheless must be mastered by the implementor himself; (2) to provide a background which will allow more ready assimilation of a text, and of even more advanced papers which will appear in the future; and (3) to provide the student with a frame of reference which should assist him in understanding the multitude of newly discovered or rediscovered techniques in the field of computer software.

Pilot Properties

Although the history of the Pilot language can actually be traced back through the Lockheed Missiles & Space Company to the Navy Electronics Laboratory, one might nevertheless coin the acronym, *P*urdue *I*nstructional

*L*anguage for *O*perating systems and *T*ranslators. Pilot itself has more than a dozen properties which contribute to its usefulness in the present context, each worth a sentence or two by way of explanation.

Efficiency. Pilot is an inherently efficient language, since virtually all of its features have been designed with both ease of compilation and speed of object code in mind. Many student implementors have produced compilers which would compile the complete compiler from source to target language in 5 seconds on a CDC 6500.

One pass. Since the Pilot compiler is based upon the one-pass principle, it must therefore illustrate a solution of the *forward reference* problem for those cases in which jumps to as yet uncompiled labels are encountered.

Reentrant. Both the compiler itself is reentrant (or *pure procedure*), as well as the code which it generates. This feature may be readily omitted, however, if it is not required.

Portable. The language has been implemented hundreds of times, on most of the large computers and many of the smaller ones, such as the IBM 1130 and IBM 1620.

Systems-orientation. The language includes notation for reaching core directly, both for fetches and stores, and for transfers. Both octal and hexadecimal numbers are handled. It also allows the insertion of one or more machine-language instructions after any statement. With these capabilities, it is clearly close enough to any machine on which it is implemented to assure the ability to produce efficient code. While it does not otherwise provide for bit handling, such a feature has frequently been added by a student implementor.

Simple language. The language itself has been reduced to a very basic simplicity, while retaining only those capabilities required to write, for example, a complete compiler. It includes neither floating point, literals, nor parenthetical grouping of expressions, and all variables are global. From the point of view of the user of the language, some of these restrictions may appear irksome, at which point it is well to remember that the user is intended to be the implementor as well, and that the restrictions have been introduced for the sole purpose of easing the task of understanding the implementation. As evidence of this simplicity, one may note that the Backus Normal Form specification of Pilot requires fewer than 30 definitions, of which five are devoted to the differences between octal, decimal, and hexadecimal numbers.

Minimal size. The Pilot compiler consists of approximately 250 source statements, an order of magnitude fewer than the compiler from which it was reduced. This fact, however, does not imply that the important features of a compiler have been eliminated. One binary order of magnitude was obtained by not including diagnostics, and another by avoiding all but the most trivial form of optimization. While it is true, and must frequently be emphasized, that half of the code in a good compiler should be concerned with error detection, this phase

is so highly dependent upon the source language chosen that the amount the student could learn in this area would not warrant the added complexity which it would introduce at this point. While some optimization techniques now have more generality than do diagnostics, they still tend to vary greatly with the architecture of the machine for which they are designed. Since the purpose requires that the compiler be readily implementable upon a wide variety of unspecified computers, the only optimization retained in the compiler is the provision that a single working register not be reloaded when it already contains the desired variable.

Crutch coding. The ability of the compiler to accept a mnemonic or numeric machine-language instruction after any statement is called *crutch coding,* implying the descent to a lower, less intelligible level of language. While this feature is not used in the compiler itself, it is essential in the implementation of the operating system, where efficiency could not be achieved without it. It is worth noting that the crutch-coding generator alone, in the absence of all other generators, provides a simple one-to-one assembler capability.

Hash coding. Since the compiler employs scatter storage addressing, it demonstrates the basic features of the only generally accepted method of handling symbol tables in compilers and assemblers.

CO–NO tables. The Pilot compiler uses a *transition matrix,* called a *C*urrent *O*perator–*N*ext *O*perator (CO–NO) table for parsing; hence it demonstrates the fastest parsing technique known.

Extensible. Rather than reducing the size of the CO–NO table to correspond to the limited size of Pilot, the matrix has been deliberately left sparse. Any combination of operators preceding and following an operand may be assigned a specific meaning, resulting in a call upon an individual generator. This allows for easy extension, by supplying a new generator for any point in the matrix not previously implemented. Further, because the compiler was obtained by compressing a large working Algol-type compiler, system design problems are avoided when it is expanded.

Modular. Despite the fact that all variables are global, the compiler for Pilot is written in modular fashion. It consists of a driver and 14 subroutines. Consequently, each of these subroutines can be analyzed and understood separately.

Self-compiler. The fact that a source listing of the compiler exists in the same language which it accepts provides several advantages. By compiling the source deck once, and using the output of that compilation to compile the same source deck again, success of the second compilation can be taken as evidence of a "bug-free" compiler. While reaching this moment of truth invariably provides the thrill of accomplishment for the compiler implementor, it also paves the way for the greater thrill, since it makes it possible to make and test extensions quickly. As a result, the student who wishes to design a completely new language

facility may do so, and obtain results with a run or two. Further, it illustrates the only known method of avoiding the usual situation, in which it is expected that fast compilation implies slow execution, and, conversely, that producing highly optimized code requires long compilation times. Since any self-compiler must execute only object code which it has itself generated, it must either generate poor code slowly, or excellent code rapidly. In practice, this means that the initial version will usually be slow, but that improvements may constantly be added. This technique has been adopted in the implementation of a number of impressive production compilers, including those for extended Fortran compilers, such as LRLTRAN and Fortran IV-H.

Versatile. The versatility of the language has been demonstrated in more advanced projects. It has been used by graduate students to write a D level PL/I to Pilot translator in Pilot language. By providing a Pilot compiler for a given computer, they are able to compile their PL/I translator. Then, by feeding the output of their translator to their Pilot compiler, they are able to compile and execute PL/I programs. Other graduate students have used Pilot in thesis research, leading to demonstrations of dynamic algebra, inverse compiling, program simplification, and the reorganization of multipass programs.

Contents

Preface ix
Prologue xi

PART I. COMPILING SYSTEMS

Chapter 1
Introduction 1
Chapter 2
Structural Design of the Pilot Compiler 9
Chapter 3
The BNF Definition of a Language 12
Chapter 4
Defining an Internal Compiler Code 16
Chapter 5
The Lexical Scan 20
Chapter 6
Number Conversion 27
Chapter 7
Hashing Symbol Tables 31
Chapter 8
The Basic Scan 36
Chapter 9
Compiling Declarative Statements 39
Chapter 10
Handling Forward References 45
Chapter 11
Compiling with a CO - NO Table 49
Chapter 12
Generating Reentrant Code 58

be inadequate for expansion into a complete software system, since it was obtained by reducing such a system.

The approach used in this manual, which has enjoyed some measure of success for several hundred students, therefore consists of the presentation of a small, systems-oriented language called Pilot, followed by two completed examples of its use, first to write a compiler, and then to write a time-sharing operating system. In a one-semester course, 85 percent of the students have succeeded in implementing the compiler with from one to fifteen extensions, while covering the bulk of the material in a text such as that of Rosen, Gries, or Donovan.

A list of those texts currently available is given in the references at the end of Chapter 1.

Lafayette, Indiana

Maurice H. Halstead

Chapter 1

Introduction

Since virtually all compilers have much in common, it should be possible to study any good compiler for a language such as Fortran, Algol, Cobol, Jovial or PL/I, and from a detailed analysis of that one compiler on a single machine, to obtain the knowledge which could be readily transferred to or from the others. While possible, such a method is uneconomic in terms of effort, primarily because such compilers, in addition to their fundamental or basic components, must of necessity contain a far larger proportion of code which is concerned only with the details of their particular environment and implementation. With that approach, the trees obscure the forest.

At the other extreme, one could distill, from the population of all compilers, those principles which they exhibit in common, and study them in a completely abstract way. While this extreme holds the advantage over the other, it suffers from a paucity of detail which leaves the student with a compartmentalized knowledge which is often too fragmentary to provide a firm base from which he could design and implement a processor independently. He may recognize a forest, but miss a tree.

Rather than choosing either extreme, this laboratory manual assumes that the broad view may be obtained from an increasing number of good books on compilers and operating systems, but that, in the final analysis, the only way that a student can both know, and know that he knows, this interesting area is to study both the objectives and the details of their implementation. This manual, therefore, provides sufficient detail to enable the student to completely understand and implement both a self-compiler and a time-sharing operating system. While both the self-compiler and the operating system have been obtained by reducing larger systems by an order of magnitude, they still retain the important functions of their predecessors.

Before starting the actual implementation of the compiler illustrated in chapters 1 through 14, it is advisable to consider briefly the various classes of processors which have been developed, and to examine the gross structure of a few of the processes which will be involved.

While there is as yet no satisfactory ordering scheme for programming

systems, just as there is no rational method for classifying computer languages by comparison of their level, for the purposes of this manual the following listing should suffice.

Machine coding. The direct preparation of the numerical code of the machine, by writing in the absolute octal, hexadecimal, decimal, or even binary, representation of the computer involved. This is the true machine language.

Simple assemblers. Systems which provide the user with a one-to-one conversion from mnemonic representations of machine operation codes and addresses to machine code. Frequently this is referred to as *machine language.*

Macro assemblers. Systems which augment the capabilities of simple assemblers by allowing the definition of new mnemonic operations not included in the repertoire of instructions of the computer, achieved by a combination of such instructions. This *one-to-many* expansion provides so much more power than that available in a simple assembler that attempts are being made to reduce its customary machine dependence.

Interpreters. Systems which are virtually the same as compilers, except that instead of generating machine code for later execution, they transfer control to a stored routine for actual execution upon recognition of each statement. While interpreters fell into disuse because of their extreme slowness, they are again becoming of interest because of the systems control they provide for multiprogramming real-time systems.

Compilers. Systems which convert higher-level language on a one-to-many basis to machine code, which may be more expeditiously ordered than the original order encountered in the source language.

Cross compilers. Compilers which operate on a given computer, but generate machine code for a different computer.

Compiler-compilers. Systems which provide a mechanism and language for semiautomatic production of compilers.

Self-compilers. A term occasionally used to describe a compiler written in its own language, usually with the proviso that it is intended to be readily extendable in source language and capable of improvement in object-code efficiency.

High-level translators. Systems which translate from a given source language to an object language which in itself is another source language. To date the principle uses of this class of translator has been to translate from one version of a given language to a later version of the same language.

Decompilers. Systems which accept as input the machine language of a given computer, and translate it to a quasi-machine-independent, higher-level language. While to date no decompilers have reduced programmer intervention to zero, they have been economically successful in converting applications programs from one generation of computers to another, and in improving documentation of early machine language programs.

The next dozen chapters concern the implementation of a self-compiler, but

as can be seen from preceding definitions, this must also cover a compiler. In addition, by virtue of the inclusion of a machine language statement type in the language definition, the self-compiler will be seen to include a simple assembler.

To prepare for these exercises, we will start by considering the broad outlines of two processes, the structure of a compiler, and the process of transferring a self-compiler from one computer to another.

The basic elements of a compiler must provide for:

1. Accepting source language from an input device
2. Converting from the binary-coded-decimal (BCD) or similar input code of the device to a more efficient internal compiler code (ICC)
3. Scanning the source string to determine which actions are required
4. Generating the proper code, either in an intermediate or machine language form.

Beyond these basic functions, a compiler should also pr · e facilities for·

1. Diagnosing errors
2. Deleting redundant expressions
3. Allocating working registers
4. Optimizing and relocating object code
5. Providing various compile-time statistics

Since these additional features are left as options to the student, and only the basic functions are demonstrated in the compiler, we may accept Fig. 1.1 as an illustration of the compiling process.

From Fig. 1.1, it can be noted that some combinations found in the source string, in ICC, will result in the immediate generation of object code, while others, such as labels, will merely result in the saving of information for later use by the compiler.

Transferring a Self-compiler

Suppose one has implemented a compiler for some language L, written in the machine or assembly language of computer X, and producing object code for computer X, and assume that a compiler for the same language is needed on, and for, a new computer Y. In this case, except for the personal experience gained, there is nothing in the original X compiler that will be of much help in producing the new Y compiler. However, if the original X compiler had also been a self-compiler, then the situation would have been somewhat different, because there would have been a source listing (and card deck) of the original X

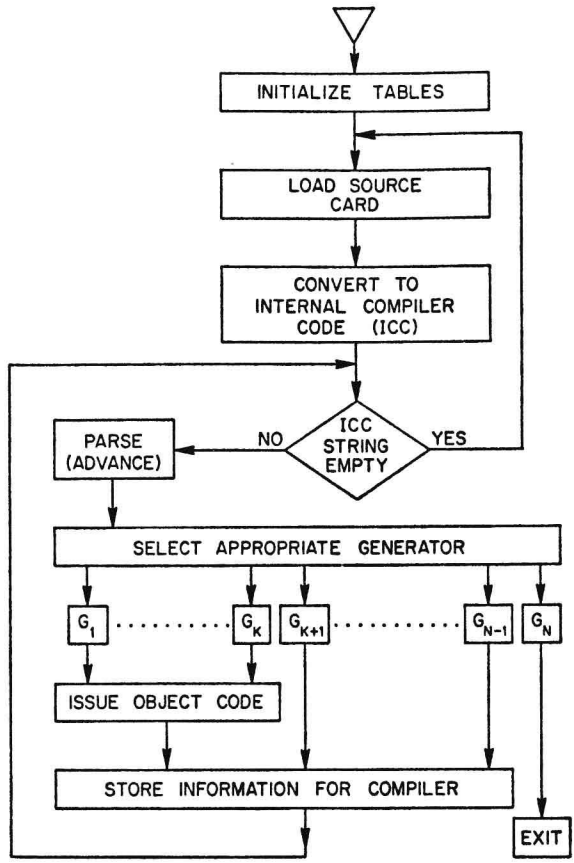

Fig. 1.1. The compiling process.

compiler in language L. Since there is to be no change in language L, none of those parts of the compiler which deal solely with the properties of language L will require any change. The only changes required in the source listing will be in: (1) the object code issued by the generators, and (2) any of the routines which depend upon the word length or character code of the machine. Let us call item 2 the *housekeeping* function, and leave it without change for the moment. If we prepare a new source deck, identical to the original, except for the generators, which we revise to make them issue the object code appropriate to the new computer Y, then we will have a source deck which can still be compiled by the original compiler. When this compilation has been performed, the result will be a new compiler, or strictly speaking, a *cross-compiler.* This cross-compiler will still run only on the original machine, X, but it will produce

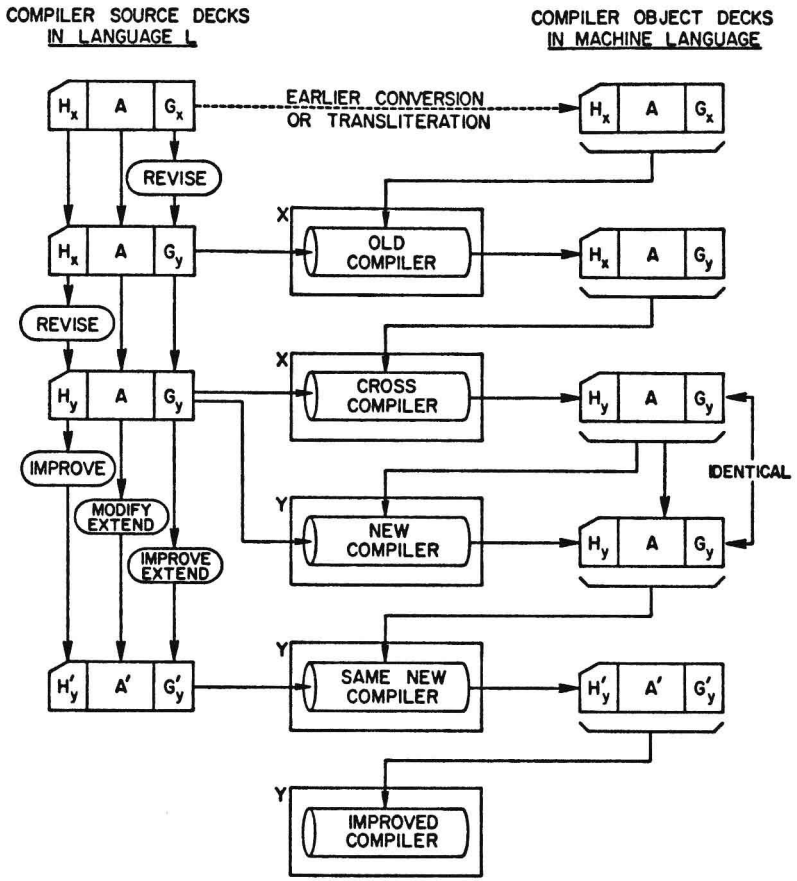

Fig. 1.2. The process of bootstrapping a self-compiler from Computer X to Computer Y.

object code which will only run on the new machine, Y. At first glance it might appear that the source listing (with its revised generators) which produced the cross-compiler could be compiled by the cross-compiler to produce a compiler which would run on, and for, the new machine. While this is almost true, in practice it is necessary to revise the housekeeping routines in the source listing before doing so.

The process is shown in Fig. 1.2, in which the symbology has the following meaning. The column on the left contains source decks of the various versions of the compilers, always written in the same language, L. Each deck contains a complete compiler, and consists of the three parts: housekeeping, **H**; analysis, **A**: and generators, **G**. The subscripts denote the machine for which the component has been specialized. Since the analysis portion is machine independent, it is not