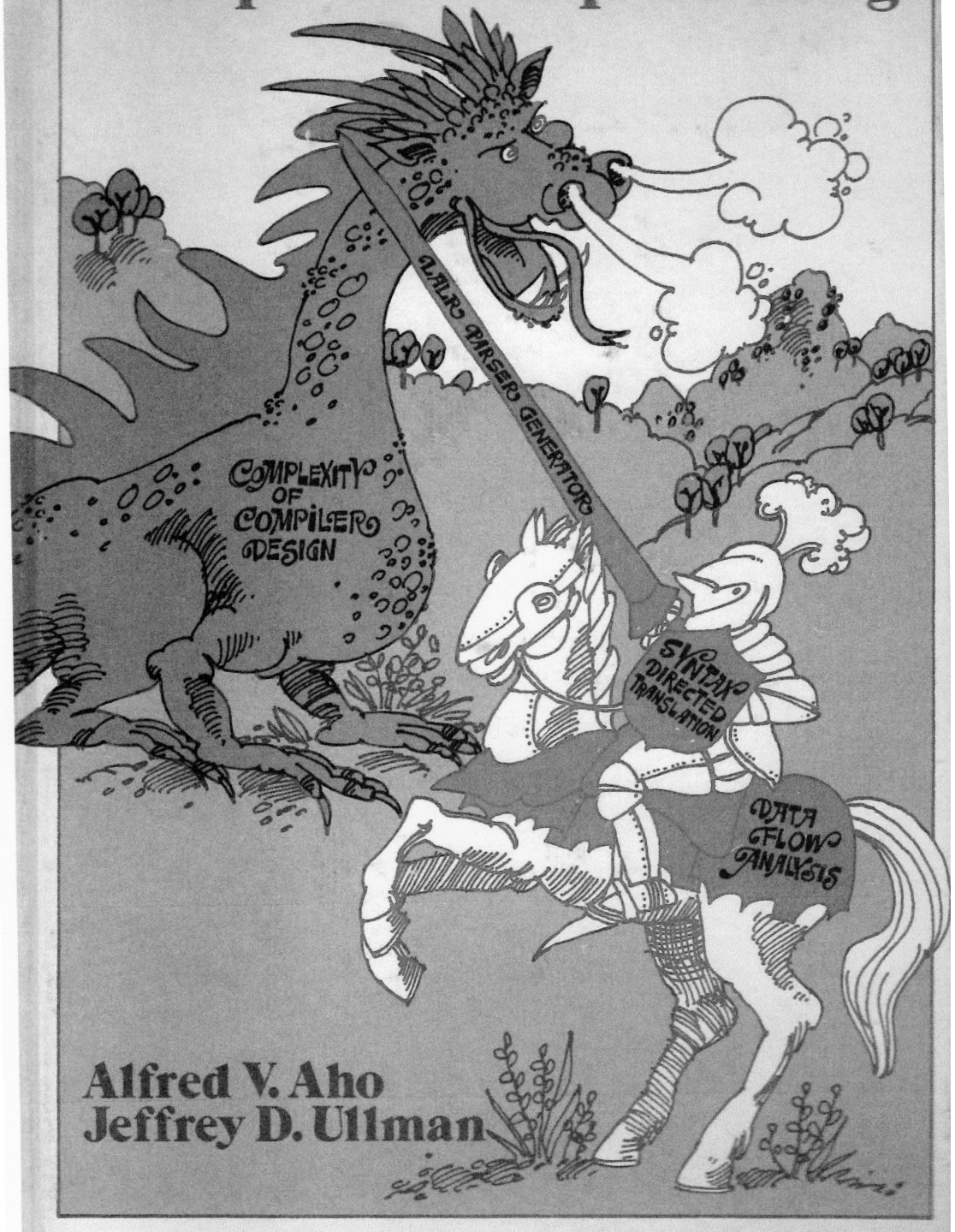
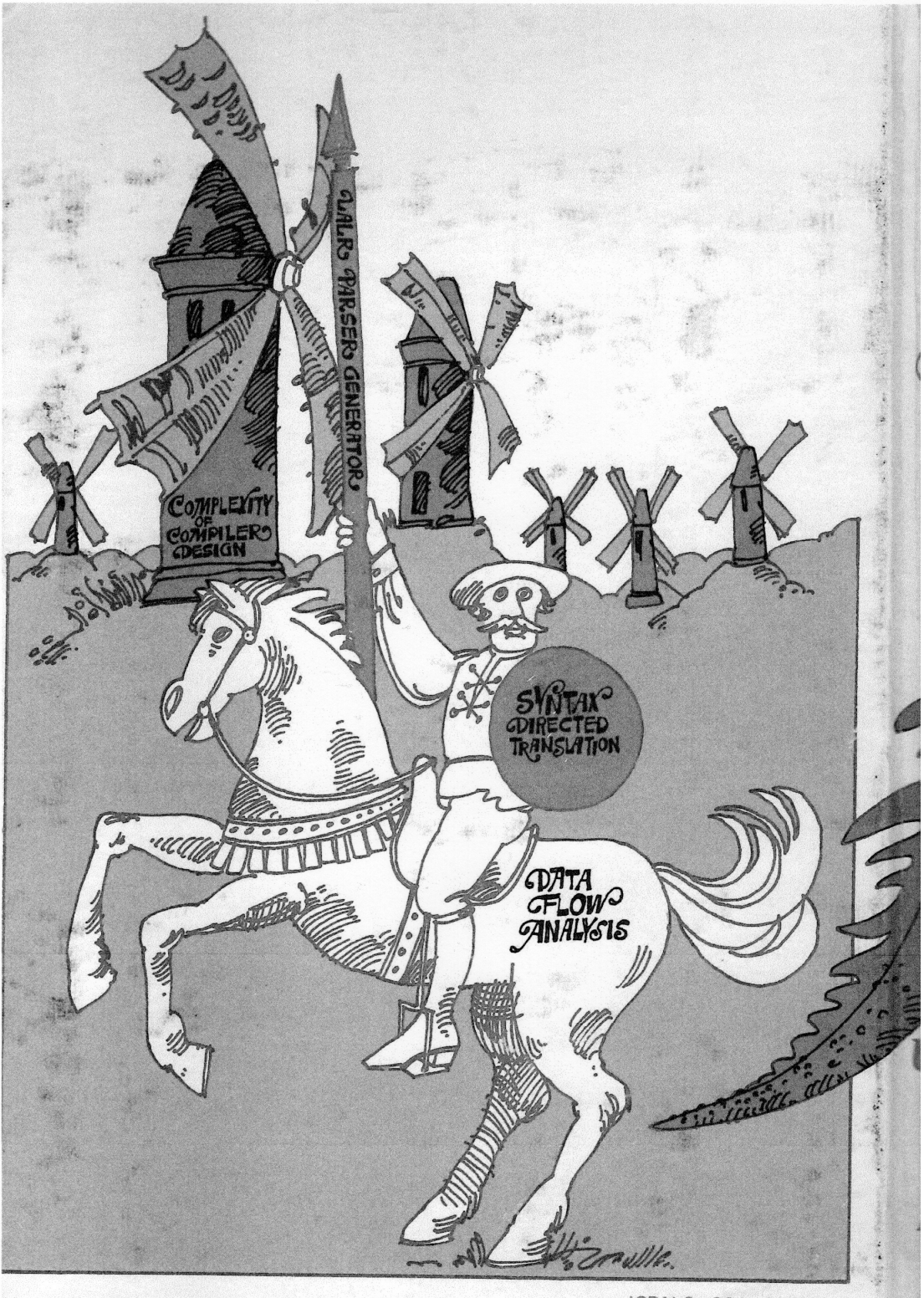
Principles of Compiler Design



Alfred V. Aho
Jeffrey D. Ullman



ISBN 0–201–00022–9

Principles of Compiler Design

ALFRED V. AHO

Bell Laboratories
Murray Hill, New Jersey

JEFFREY D. ULLMAN

Princeton University
Princeton, New Jersey

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts • Menlo Park, California
London • Amsterdam • Don Mills, Ontario • Sydney

This book is in the
ADDISON-WESLEY SERIES IN
COMPUTER SCIENCE AND INFORMATION PROCESSING

Michael A. Harrison
Consulting Editor

Second printing, March 1978

Reproduced by Addison-Wesley from camera-ready copy supplied by the authors.

Copyright © 1977 by Bell Telephone Laboratories, Incorporated. Philippines copyright 1977 by Bell Telephone Laboratories, Incorporated.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada. Library of Congress Catalog Card No. 77-73953.

ISBN 0-201-00022-9
BCDEFGHIJK-HA-798

CHAPTER 15

Cod
Generatio

We now turn to code generation, the final phase of compilation. Goo code generation is difficult, and not too much can be said without dealin with the details of particular machines. Nevertheless, the subject must b studied because a careful code-generation algorithm can easily produce cod that runs perhaps twice as fast as code produced by an ill-considered algc rithm.

We begin our study by looking at the problems that arise in generatin good code. We initially develop a straightforward algorithm to generat code from a sequence of three-address statements. We progress to mor complex algorithms that take into account the flow-of-control structure of program in assigning registers to names. We then discuss a method fo generating reasonably good code, treating a basic block or an expression a the fundamental entity. Finally, we consider various local, machine dependent optimizations that have been found effective in improving th quality of output code.

15.1 Object Programs

We assume that the input to our code-generation routine is an intermediat language program that can be a sequence of quadruples, a sequence of tri ples, a tree, or a postfix Polish string. The output of a code generator i the object program. This may take on a variety of forms: an absolut machine-language program, a relocatable machine-language program, a assembly-language program, or perhaps a program in some other program ming language.

Producing an absolute machine-language program as output has th advantage that it can be placed in a fixed location in memory and immedi ately executed. A small program can be compiled and executed in a fe seconds. A number of "student-job" compilers, such as WATFIV an PL/C, produce absolute code.

Producing a relocatable machine-language program (object module) a output allows subprograms to be compiled separately. A set of relocatabl object modules can be linked together and loaded for execution by a linkin

Ex.1049 / Page 5 of 6
Cisco Systems, Inc.

loader. Although we must pay the added expense of linking and loading if we produce relocatable object modules, we gain a great deal of flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object module. Many commercial compilers produce relocatable object modules.

Producing an assembly-language program as output makes the process of code generation somewhat easier. We can generate symbolic instructions and use the macro facilities of the assembler to help generate code. The price to be paid is the assembly step after code generation. Nonetheless, since by producing assembly code the compiler will not duplicate the entire task of the assembler, this choice is another reasonable alternative, especially for a machine with a small memory, where a compiler must use several passes.

Producing a high-level language program as output simplifies code generation even further. For example, FORTRAN is the output of the ALTRAN† compiler (Brown [1977]) as well as a variety of "Structured FORTRAN" preprocessors such as RATFOR (Kernighan [1975]). Generating FORTRAN as output defers the problems of machine code generation to the FORTRAN compiler, yet allows the compilation of languages that extend FORTRAN in useful ways, as in the cited examples.

The Environment of the Code Generator

We assume that the code generator is presented the intermediate text in one of the forms discussed in Chapter 7; we shall, in our specific algorithms, deal with quadruples or in one case, with parse trees as the intermediate form. We assume that all necessary semantic checking has been done, either during intermediate code generation, in a separate pass, or just before each quadruple is examined by the code generator. Thus the data type of each operator and operand is known, and all necessary type conversions are present in the intermediate text. It is also assumed that data areas and offsets have been determined for each name, as discussed in Chapter 10, and that this information is present in the symbol table.

It is often helpful to imagine that the intermediate text has been partitioned into basic blocks, as in Algorithm 12.1, but with the additional proviso that procedure calls are isolated in blocks by themselves. It is not essential that a data structure representing a flow graph be present. Algorithm 12.1 tells us how to recognize the beginning and end of a basic block as we scan a list of quadruples. Thus our algorithms for code generation are in no way predicated on the existence of an optimization phase in the compiler.

---

† A language for algebraic manipulation.