An Introduction to Compiler Writing

J. S. Rohl
University of Manchester Institute of Science and Technology

Macdonald and Jane's · London and
American Elsevier Inc. · New York

Sole distributors for the United States and Dependencies and Canada:
American Elsevier Publishing Co. Inc., 52 Vanderbilt Avenue, New York, N.Y. 10017

Sole distributors for the Continent of Europe excluding the British Isles and Commonwealth and the Republic of Ireland:
Elsevier Publishing Company, P.O. Box 211, Jan Van Galenstraat 335, Amsterdam, The Netherlands

Sole distributors for all remaining areas:
Macdonald and Jane's, Macdonald & Co. (Publishers) Ltd.,
Paulton House, 8 Shepherdess Walk, London, NI7LW

Macdonald ISBN 0 356 08173 7
American Elsevier ISBN 0 444 19523 8
Library of Congress Catalog Card No. 74 16914

Text set in 10/12 pt. Monotype Times New Roman, printed by letterpress and bound in Great Britain at The Pitman Press, Bath

Preface

This is, as its title suggests, an introduction to compiler writing. It is written in a narrative style. We start with a basic language and show how to compile it; and then expand the language by degrees showing how the compiler can be modified to accommodate these expansions. Thus, the description of a technique might be enhanced from chapter to chapter. Since the information is not collated (except through the index) the book's use as a work of reference is, to that extent, limited.

Since it is an introductory book, the number of language facilities that we consider is restricted. The basic language expands to about the full power of Algol, but not to, say, Algol 68. We assume a working knowledge of Algol and, in so far as Fortran is the classical representative of the class of languages which can be implemented with a static storage allocation scheme, we assume a knowledge of it, too.

We are concerned with the broad principles rather than detailed representation. Accordingly, language facilities described have sometimes been modified to avoid some of the well-known minor problems of implementation. We do not consider jumping out of blocks, for example, or own arrays. The order code of the target machine has been carefully chosen to reflect the structure of the source language. It is, in fact, the order code of MU5 (see ref. 0.1) and this book might serve to illuminate the philosophy on which the order code of that machine is based. Since the order code of ICL's New Range, the 2900 Series, derives from that of MU5 it will apply to that series as well, except that some concepts (in particular the Boolean facilities) are not relevant.

All books reflect their author's views on his subject. Three views have shaped this book.

First, that computer science has become of age to the extent that its structure as a discipline is now clearer.

There are areas such as program design, data structures and so on in which a coherent body of knowledge now exists. I have assumed that readers of this book will be well versed in the program structures of Algol (in particular the recursive calls of procedures and the while clause) and in data structures (particularly binary and general trees).

Second, and almost as a corollary, that the theory of grammars is a subject in its own right rather than a part of compiler writing. The compiler writer will see it as a tool, of course, in the same way as he sees data structure techniques, and will use it as such. Most other books on compiler writing treat the theory of grammars as central to the whole subject. This is, perhaps, the major difference between this book and those: here the subject is introduced at the point at which it is relevant, views from both ends of the spectrum (the source-driven precedence technique and the syntax-driven, top-down analysis technique) are given, and references made to the theory for those wishing to refine the techniques. If I had to put in one sentence what I feel compiler writing is about, I would say that it is about the structures which have to be maintained by the compiler and the procedures by which they are created and transformed.

Third, that the compiler writer has an important role in the design of computer systems as an interpreter, both of the users' needs to the machine designer, and of the machine's capabilities to the language designer. Consequently, throughout the book, I make references to the curious constructions still existing in the languages in use today and to the lines along which hardware might develop to make it more amenable to the requirements of high-level languages.

There are a restricted set of references in this book. I have tried to be selective, referring only to those papers and books that a student might be expected to read. Such a choice is, of necessity, a personal one. For those students who are fired with enthusiasm the references themselves contain the further references.

This book has had a long gestation period. It started with some post-graduate lectures I gave in 1967 in the University of Queensland and I am grateful to Professor S. A. Prentice for giving me the opportunity of developing my ideas of teaching the subject along these lines. Since then, of course, they have been refined many times in the course of presenting them to many sets of students, and I should like to thank all those who, wittingly or unwittingly, have helped me comb out a number of errors. In its draft form this book has been read by a number of people and I should like to record my appreciation of all the comments, corrections and suggestions I have received. I would particularly single out my research students, Graham White and Alan Brook, and Professor Gordon Rose.

Finally, I should like to record my indebtedness to Hilary Mayor, who, as Hilary Shaw, typed and retyped my many attempts to get the early chapters to say what I wanted to say and who drew all the diagrams in those chapters; and to Susan Green who typed subsequent chapters, drew the diagrams they contained and undertook the quite substantial revisions of the whole book.

Contents

1 Introduction

What is a compiler? Conceptually at least, a compiler is a program just like any other program. If we consider Fig. 1.1 (i), a program is something which reads in some data and produces some results.

A compiler is a program which takes as data the program being compiled (called the *source program*), and produces as its results, an equivalent program in binary machine code (called the *object program*), as shown in Fig. 1.1 (ii). This object program is of course

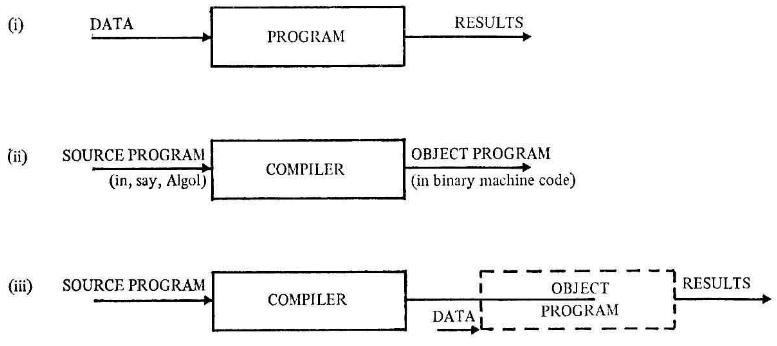

Fig. 1.1. The structure of (i) a program; (ii) a compiler; (iii) a compiler and its compiled program.

a program, so it then usually reads in some conventional data and prints out some results, as shown in Fig. 1.1 (iii).

We have said that this is conceptually what happens, and in many cases it is precisely what happens. This is the simplest form of compiling system, called a *compile-and-go* system, and we will assume it throughout the main part of this book. In Chapter 15, however, we will consider other systems.

Most programs these days are written in a high-level language such as Fortran, Algol, Cobol, PL/1 and so on. (Hence the need for compilers.) The same is becoming true of compilers: sometimes Fortran, Algol or PL/1, sometimes in languages specially designed for the purpose. When we wish to illustrate some point we will use Algol, augmented if necessary; in Chapter 16 we will discuss the problem in more detail.

2 A compiler for a basic language

Let us approach the basic problems of compiler writing by considering a very *basic language* and a compiler for it.

2.1 The basic language

Consider a very small sub-set of Algol which results from the following restrictions:

(i) There are no blocks, compound statements or procedures.
(ii) Only one 'instruction' is permitted per line.
(iii) Identifiers (including those for labels) consist of single letters.
(iv) All real constants contain a decimal point; no integer constants do.
(v) Arrays are real, and of one dimension only (they are vectors); the lower bound is always 0, and the upper bound is always a constant, as shown by the declaration:

array $p[0:10]$

Further, the bounds of each individual array must be given explicitly.
(vi) Expressions are very simple in that they contain only one or two operands, that the operators available are $+$, $-$, $*$, $/$ (where $+$ and $-$ are always binary, never unary) and that both operands (where there are two) must be of the same type. For example:

$h * i$
0
$j + 1$

(vii) In an assignment statement the left-hand side variable must be of the same type as the right-hand side expression.

(viii) The go to statement can refer only to a simple label as in:

go to l

There are no switches.

(ix) The only statement that can be made conditional is the go to statement. For example:

if $j \neq n$ **then go to** l

(x) The only form of Boolean expression is the relation between two operands of the same type.

(xi) There are no built-in functions.

(xii) All programs end with a dummy statement.

These restrictions are designed to make the language as simple as possible while still retaining the essential characteristics of a high-level language.

To fix ideas, let us consider a program to calculate n integrals using Simpson's Rule:

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{3}\,[f0 + 4f1 + f2]$$

where $h = (b - a)/2, f0 = f(a), f1 = f\left(\frac{a+b}{2}\right), f2 = f(\mathrm{b})$.

The values of $h, f0, f1$ and $f2$ are given as data for each integral. If we add some rudimentary input and output statements for completeness we arrive at a program such as:

```
    begin
    real h, e, f, g, i;
    integer j, n;
    j := 0;
    read n;
 l: read h, e, f, g;
    i := 4·0*f;
    i := e + i;
    i := i + g;
    i := h*i;
    i := i/3·0;
    print i;
    j := j + 1;
    if j ≠ n then go to l;
    end
```