

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

© J. S. Rohl 1975

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

**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 prior permission of
the publishers.**

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

Preface

1 Introduction	1
2 A compiler for a basic language	3
2.1 The basic language	3
2.2 A compiler for this language	5
2.3 Read a line	7
2.4 Process a line	8
2.5 Modular hierarchical structure	8
2.6 Faults	8
2.7 Information needed by the compiler	10
2.8 Declarations	11
2.9 Assignments	13
2.10 Efficiency	18
2.11 Labels and go to statements	21
2.12 Conditional go to statements	26
2.13 Input and output	29
3 Multi-character identifiers and lexical analysis	30
3.1 Multi-character identifiers	30
3.2 Hashing	35
3.3 Lexical analysis	36
3.4 Lexical analysis of constants	41
3.5 Backtracking	41
3.6 Punching conventions	43
3.7 Comments and strings	47
3.8 Other actions	48
3.9 The relevance to Fortran	48
4 Expressions and syntactical analysis	50
4.1 Expressions	50
4.2 A revised over-all flow diagram	51
	xi

4.3	Equivalent simple assignments	53
4.4	Re-ordering simple assignments	56
4.5	A finally revised flow diagram	59
4.6	Re-ordering in Algol	59
4.7	Binary trees	62
4.8	Code generation from a binary tree	64
4.9	Allocation of temporary variables	66
4.10	Transformation of a binary tree	66
4.11	A precedence algorithm	68
4.12	Unary operators	69
5	Reverse Polish notation	73
5.1	Reverse Polish	74
5.2	Conversion to Polish notation	74
5.3	Code generation for a zero-address machine from Reverse Polish	76
5.4	Code generation for a one-address machine from Reverse Polish	78
5.5	Code generation for a one-address machine from Reverse Polish allowing re-ordering	79
5.6	Re-ordering a Reverse Polish string	81
5.7	The relationship between binary trees and Reverse Polish strings.	82
6	A general tree structure	84
6.1	N-ary trees	85
6.2	Creation of an n-ary tree	87
6.3	Code generation from an n-ary tree	88
6.4	Transformation of an n-ary tree	89
6.5	Unary operators	91
6.6	N-ary trees and Algol	92
6.7	An assessment of n-ary trees	93
7	Booleans and conditions	94
7.1	Boolean expressions	94
7.2	Internal representation	95
7.3	Code generation from a binary tree with no relations	95
7.4	Code generation from a binary tree with relations	97
7.5	Conditional statements	101

7.6	An alternative encoding	103
7.7	The conditional approach	106
7.8	An appraisal of the approaches	112
7.9	The conditional approach and Algol	113
7.10	Conditional expressions	114
8	Mixed expressions and assignments	117
8.1	Explicit type conversion	117
8.2	Implicit type conversion	120
8.3	Constants	122
8.4	Assignments	122
8.5	Types of expression	124
8.6	Further types of variable	125
8.7	Multiple assignments	127
8.8	A comment on this chapter	129
9	Optimisation	130
9.1	Elimination of unnecessary store and load orders	131
9.2	Evaluation of constant expressions at compile time	132
9.3	Elimination of common sub-expressions	135
9.4	Are these simple optimisations worth it?	141
9.5	Remembering the contents of registers within statements	143
9.6	Remembering the contents of registers across statements	144
9.7	Common sub-expressions across consecutive statements	144
9.8	Re-ordering statements	145
9.9	N-ary trees again	147
9.10	Loops	150
9.11	Optimisation over simple loops	151
9.12	Compound statements and optimisation over larger loops	152
9.13	Is this more general optimisation worth it?	155
9.14	Delaying the storage of registers	156
9.15	Recognition of special cases	157
9.16	Machine-dependent optimisation	158
9.17	The place of optimisation	161

10 Arrays	163
10.1 Static vectors	163
10.2 Checking vector accesses	166
10.3 Static matrices	168
10.4 Optimisation of matrix accesses	169
10.5 Checking matrix accesses	173
10.6 Dynamic vectors	175
10.7 Dynamic matrices	178
10.8 Iliffe vectors	179
10.9 Three and higher dimensional arrays	182
10.10 A comparison of Iliffe vectors and the multiplication method	183
10.11 Storing a multiplication vector	184
10.12 A comment on this chapter	185
 11 Block structure	 186
11.1 Blocks	186
11.2 Blocks in a language with static storage allocation	187
11.3 Blocks in a language with dynamic storage allocation	188
11.4 Storing temporary variables	192
11.5 Coalescing blocks	193
11.6 Separating static from dynamic variables	194
11.7 Compile time considerations	196
 12 Procedures	 199
12.1 Procedure or subroutine structure	199
12.2 Compile time considerations	200
12.3 Run time considerations	201
12.4 Procedures in a language with static storage allocation	201
12.5 Procedures in a language with dynamic storage allocation	203
12.6 Parameters	206
12.7 Parameters called by value	206
12.8 Parameters called by reference	208
12.9 Optimisation of parameters called by reference	209
12.10 Parameters called by substitution	210

12.11	Optimisation of parameters called by substitution	214
12.12	Other types of parameter	215
12.13	Functions	217
12.14	Recursion	218
12.15	Some further compile time considerations	220
12.16	An alternative strategy	221
13	Full syntax analysis	224
13.1	Backus Naur Form	224
13.2	Syntax trees	225
13.3	A modified set of definitions	226
13.4	Processing syntax trees	229
13.5	An analysis or parsing algorithm	229
13.6	Representation of phrase definitions	231
13.7	Representation of the syntax tree	232
13.8	An analysis procedure	235
13.9	Merging alternatives	237
13.10	Phrase routines	238
13.11	Syntax analysis and lexical analysis	240
13.12	Syntax analysis and precedence	240
13.13	The theory of grammars	241
14	Fault monitoring	243
14.1	Compile time errors	244
14.2	Faults detected while converting to a standard form	246
14.3	Faults detected during lexical analysis	248
14.4	Faults detected during syntactical analysis	250
14.5	Faults detected during semantic processing	251
14.6	Other information printed at compile time	252
14.7	The lay-out of the compile time monitor	253
14.8	Run time monitoring	254
14.9	Post-mortem dumps and stack interpretation	255
14.10	Fault recovery	258
14.11	Tracing	258
14.12	A comparison between stack interpretation and tracing	259
14.13	When a program just gives the wrong results	259
14.14	The user's options	260

15 Compiling systems	262
15.1 Semi-compiled format and relocatable binary	262
15.2 Libraries with a compile-and-go compiler	265
15.3 In-core batching compilers	266
15.4 Cafeteria systems	268
15.5 Multi-pass compilers	269
15.6 Interaction: re-entrant (or pure) code	270
15.7 Interaction: incremental compiling	274
15.8 Interpreters	278
15.9 The future	280
 16 Compiler writing aids	 281
16.1 Control structures in a compiler writing language	281
16.2 Data structures in a compiler writing language	282
16.3 Fault monitoring in a compiler writing language	284
16.4 Special procedures for compiler writing	284
16.5 Compiler target languages	285
16.6 Generators	286
16.7 Boot-strapping	287
 17 Designing compilers	 289
17.1 Choice of compiling system	289
17.2 Choice of compiler writing aids	290
17.3 Choice of mapping between source language and object machine	291
17.4 Choice of algorithms and data structures	292
17.5 Encoding and testing the algorithms	293
17.6 The future	294
 A 1 The basic language used in Chapter 2	 295
A1.1 A line-by-line definition	295
A1.2 A definition of a complete program	296
 A 2 The order code of the object machine	 297
A2.1 The arithmetic orders	297
A2.2 The Boolean orders	298
A2.3 The comparison orders	298

A2.4	The control orders	298
A2.5	The descriptor operations	299
A2.6	The orders concerned with SB and SF	299
A 3	Line reconstruction	300
	References	303
	Index	305

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

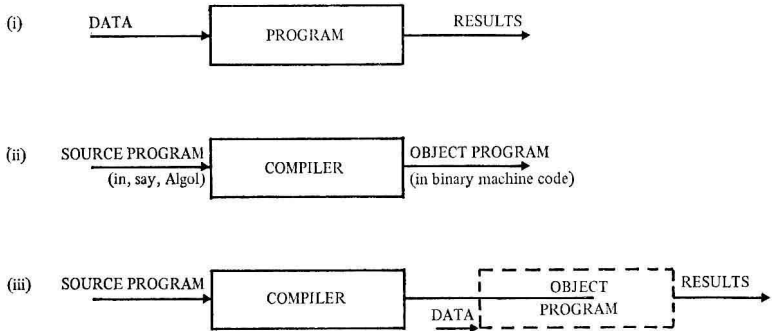


Fig. 1.1. The structure of (i) a program; (ii) a compiler; (iii) a compiler and its compiled program.

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

AN INTRODUCTION TO COMPILER WRITING

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:

$$\int_a^b f(x) dx \approx \frac{h}{3} [f_0 + 4f_1 + f_2]$$

where $h = (b - a)/2$, $f_0 = f(a)$, $f_1 = f\left(\frac{a+b}{2}\right)$, $f_2 = f(b)$.

The values of h , f_0 , f_1 and f_2 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  $j \neq n$  then go to  $l$ ;
end
```