RECURSIVE DESCENT COMPILING

A. J. T. DAVIE, B.Sc.

RECURSIVE DESCENT COMPILING

A. J. T. DAVIE, B.Sc.
and R. MORRISON, B.Sc., M.Sc., Ph.D.
Department of Computational Science
University of St. Andrews
Scotland

ELLIS HORWOOD LIMITED
Publishers · Chichester

Halsted Press: a division of
JOHN WILEY & SONS
New York · Brisbane · Chichester · Toronto

First published in 1981 by

ELLIS HORWOOD LIMITED
Market Cross House, Cooper Street, Chichester, West Sussex, PO19 1EB, England

The publisher's colophon is reproduced from James Gillison's drawing of the ancient Market Cross, Chichester.

Distributors:

Australia, New Zealand, South-east Asia:
Jacaranda-Wiley Ltd., Jacaranda Press,
JOHN WILEY & SONS INC.,
G.P.O. Box 859, Brisbane, Queensland 40001, Australia

Canada:
JOHN WILEY & SONS CANADA LIMITED
22 Worcester Road, Rexdale, Ontario, Canada.

Europe, Africa:
JOHN WILEY & SONS LIMITED
Baffins Lane, Chichester, West Sussex, England.

North and South America and the rest of the world:
Halsted Press: a division of
JOHN WILEY & SONS
605 Third Avenue, New York, N.Y. 10016, U.S.A.

British Library Cataloguing in Publication Data
Davie, A. J. T.
Recursive descent compiling –
(The Ellis Horwood series in computers and their applications)
1. Compiling (Electronic computers)
2. Electronic digital computers
I. Title II. Morrison, R.
001.64'25 QA76.6

Library of Congress Card No. 81-6778 AACR2

ISBN 0-85312-386-1 (Ellis Horwood Limited)
ISBN 0-470-27270-8 (Halsted Press)

Typeset in Press Roman by Ellis Horwood Limited
Printed in England by R. J. Acford, Chichester

Table of Contents

Author's Preface

The computing community is well served with texts, good, bad and indifferent, on the subjects of compiling and compilers. One of the most popular methods of implementing a compiler is that of recursive descent and many compilers, including ones for the languages Algol 60, Pascal, Algol 68R and BCPL, have been written using this technique. It is therefore surprising that comparatively little has been written about it. This text sets out to bridge this unexpected gap.

The subject matter of the text has formed the basis of lecture courses at St.Andrews University for both undergraduates and graduates. Naturally the courses have developed, and will continue to do so, over the years. At present the material appears in, but does not form the whole body of, lectures on Graph Theory, Grammars and Automata and Compiling Techniques.

The text sets out to give an introductory look at compiling in general through the medium of one particular technique. It does not therefore claim to be a complete reference guide to all aspects of compiling. Several topics, for instance that of optimization, are only touched on briefly. The intention has been to set out the main problems encountered in any compiler, however simple, and show how to tackle each of these in the relatively straightforward way which recursive descent imposes.

Students who embark on any course on compilers will be expected to know something about programming languages. This text assumes that the readers have such knowledge, that they will be fairly proficient at programming computers and that they will know about the fundamentals of program and data structures and how to manipulate them. It does not assume knowledge of any *particular* programming language. However familiarity with a block or procedure structured language such as one of the Algols or Pascal would be an advantage. As far as elementary mathematics is concerned, only basic set theory and logic are assumed as prerequisites.

The book is divided roughly into three parts. The first, which consists of Chapter 1 by itself, is a general introduction. The second, comprising Chapters 2, 3 and 4, is mainly theoretical in nature. Chapter 2 introduces some essential

mathematical notions chiefly that of closure. These are used in Chapters 3 and 4 which are about linguistic specification and testing whether languages are suitable for the recursive descent treatment. Some hints are also given about how to massage a linguistic specification into the correct form.

The third and major part of the book is about the practical realisation of a recursive descent compiler. This is done by specifying a syntax-recognising skeleton and adding flesh and muscle to it layer by layer. Chapter 5 gives an overview of this process and outlines the different layers. Chapter 6 describes the skeleton in detail and Chapter 7 the lexical analysis phase. Layers dealing with errors and types are added in Chapters 8 and 9 and ideas of scope and naming are discussed in Chapter 10. Chapters 11 and 12 deal with code generation, the former with what code is generated and the latter with how to generate it. The final chapter stands on its own. It is, strictly speaking, not specific to our particular kind of compiler; but we felt that the subjects of bootstrapping and portability were too important to be left out of any book about compilers.

We are indebted to many for helping with this book both directly and indirectly: To many of our Honours students for reading and proofreading early and later versions of various chapters and, by so doing, revising for their examinations: To our colleagues, especially Pete Bailey and Iain Adamson, for many useful comments, suggestions and criticisms: To our former colleague, Dave Turner, for his expertise and influence on our views about compilers: To our wives who have put up with it all and provided nourishment and encouragement. We must also show our gratitude to our children who, in spite of the fact that they have actively hindered this book's production, have amused and entertained us by way of diversion.

Tony Davie and Ron Morrison
St.Andrews
May 1981

CHAPTER 1

Introduction

1.1 WHAT ARE COMPILERS?

A **compiler** is a computer program which translates another program called the source **program** into yet a third called the **object program.** The source programs are written in the **source language** and each solves a particular problem for a user; the object program produced for it solves the same problem but is expressed in the **object language.** In general, the source language should be one in which users find it easy and natural to solve their problems, and the object language, whilst probably quite opaque in meaning to users, will be a natural one for some machine to execute. Thus we can view a compiler as a tool which transforms programs from the users' domain of problem solving into the machine's domain of problem execution, without varying the **semantics,** (i.e. meanings) of the programs.

It will be well known to all programmers that programs normally pass through several stages of development. Let us summarise them here. Firstly they are **created,** initially in the users' minds and then in some computer in source language, probably using an editor. They are then compiled into object language. This stage is known as **compile time** when the program is scanned, perhaps several times, to discover its static or lexicographic properties. Compile time errors may be reported, in which case the editor will be reinvoked to change the erroneous program to the intended one; or, if the user has been more skilful, an object program may be produced. This may be stored away in the computer's file system for subsequent combination with other compiled programs (such as library routines) during **load time.** When it has been loaded the combined package will be executed during what is known as **run time.** Alternatively the object program, if self contained (i.e. only containing reference to standard facilities – not other user defined routines) may miss out the load stage if the compiler is set up to place the object program straight into store ready for execution. Such a combination, where compile time and run time are run into one another is called a **compile and go compiler.**

Run time may take the form of the computer *directly* executing the object

program if it is suitable; alternatively, it may consist of the computer **interpreting** the object code. It is sometimes convenient that the object language be different from the 'native' language of any particular computer. Many source languages' philosophies suggest very forcibly the architecture of a 'natural' machine for many to run on and the object language will be the machine code for this hypothetical machine. It is rare for the architecture underlying a source language to match that of a real computer because, sadly, hardware designers and language designers very seldom get together at the start. An impressive exception is the Burroughs 5000 and 6000 [1] series computer range where the architecture was designed to support Burroughs' own version of Algol 60 [2]. If the architectures of the real and hypothetical machine don't match we have two alternatives: we can either, as mentioned already, interpret the object program by the process whereby the real machine simulates the hypothetical one or we can pass the object code through another translation stage to turn the 'natural' machine code into code for the real computer. It can be mentioned in passing that there are interpreters which *directly* interpret source code for some languages without any compile time at all (e.g. APL [3]) but these will not concern us here.

During run time, whether interpretative or otherwise, the dynamic execution of the program takes place. At this stage we may either get run time errors or alternatively correct results may be obtained. In the former case the edit-compile-load-run cycle will have to be reinvoked; even in the latter case, it may be reinvoked if the 'correct' results are the answer to a problem which is different from the one the user intended, or if he wants to modify the program in the light of the results.

To summarise, the two main stages are compile time and run time, during which static and dynamic scanning of the programs take place respectively. Many of the interesting problems of language design and compilers become apparent when we try to separate the static aspects of the language under consideration from the dynamic ones. Can the compiler tell statically whether or not a variable name has been declared for a given usage (i.e. whether it is in scope)? This is the case for most Algol-like languages but not for LISP-like languages [4]. Can it tell what type a variable has? Can it tell what value an identifier has? Can it even tell if it is guaranteed to have *some* value?

1.2 THE PHASES AND PASSES OF A COMPILER

It is an opinion almost universally held throughout the computing community that we should think about the problems we want to solve in a modular way; that is, we should try to break down complicated tasks into easier subtasks which in turn get broken into yet simpler problems until we arrive at ones which are 'trivial' to solve.

How can we break up the process of compiling into subtasks? We shall confine ourselves in this chapter to the top level of such a refinement. The

top-level subtasks we give here are common to most compilers and are known as **phases**. Later we shall see how each of the main phases breaks up into lower level subtasks.

The compiler must analyse the source program and synthesize the object program. In fact nearly all compilers perform the analysis in two distinct phases called lexical and syntactic analysis.

Lexical Analysis

The lexical phase consists of an analysis of the **microsyntax** of the source program. By analogy with spoken or written languages, this involves the collecting together of phonemes or letters to form words without any reference to the relationship of the words to one another, or to their meaning. In computer languages it means the processing of a string of characters, transforming them into a string of **basic symbols**. These include keywords (or reserved words) such as 'if', 'begin' and 'write', single symbol punctuation marks such as "(","]" and ",", operator symbols such as "+" and "*", multiple symbols of the two above kinds such as "<=", ".LE." and "::", assembly of **literals** such as "1", "3.7" and "true", and finally the collecting together of the characters in identifiers such as "x" and "mean.temperature". Writing the lexical analysis phase is not always trivial. Consider the Fortran statements:

```
DO 1 I = 1,12
DO 1 I = 1.12
```

and

```
IF(I(J) - I(K))1,2,3
IF(I(J) - I(K)) = 123
```

Are 'IF' and 'DO' keywords? In the first example 'DO' is a keyword. In the second 'DO1I' is a variable name because Fortran insists that blanks are non-significant. In the third 'IF' is a keyword, and in the last it is the name of an array being subscripted. It doesn't make the problem any easier that the subscript can be arbitrarily long and complicated.

Syntax Analysis

The second analysis phase is syntax analysis. Again by analogy with 'human' languages, this corresponds to such actions as finding the verb, subject and predicate in sentences and parsing them. In computing terms, some of the actions the syntax analyser takes are: check that in Algol like languages the *begins* and *ends* match up; make sure in Fortran that a DO statement referencing a label actually finds a statement with that label later on; and that DO loops do not overlap. The input to this phase is the string of basic symbols produced by the lexical analysis. What is the output? We shall see later that it is a **parse tree** which is an internal form of the program in a structure which allows subsequent

phases to see the relationship of the parts of the program to each other and to the whole program.

Code Generation

The third important phase, that of **code generation** is synthetic rather than analytic. It takes the parse tree and traverses it. Based on the structural relationships it finds there it produces object code in at least a preliminary form.

We could stop at these three phases because they are all common to virtually every compiler but we will mention here some other optional ones. Some compilers [5] have a **prepass** phase which does some macro expansion, allowing the user the facility of making contractions of commonly used phrases in his program. We have already mentioned that a further translation phase may be written after code generation in order to convert 'hypothetical' to 'real' machine code. This too is sometimes accomplished by macro expansion. The third and last optional phase we shall mention is that of **optimization**. In environments where large programs go into heavy production use it will be advantageous to make the object programs produced as efficient in time (or possibly space) as possible. One way of solving this problem is to measure the program at run time in order to find out which parts of the program take the longest time and to hand code these sections in assembly language. A good optimizer should do this automatically. Note however a fundamental conflict; how can a compiler which only sees the static aspects of the source program measure the dynamic performance of the object program?

Optimization phases can occur at any stage of compilation: at the beginning where it is called **global** optimization and often means automatic rewriting of source code (e.g. taking unnecessary commands out of loops); in between two other phases (e.g. to optimize the tree produced by the syntax analyser); or right at the end to improve the code produced by the code generator.

The organisation of the phases built into a compiler can be one of a number of kinds. In particular, one decision the compiler writer has to take is how to organise the phases into **passes**. A multipass compiler makes complete scans over the various forms the program goes through, both internal and external. Each pass reads the output from the previous one (or the source program if it is the first pass) and produces complete output for the next pass. No pass will be invoked until the previous one is complete. For example if we were to organise the lexical analysis phase as a pass, the compiler would first completely scan the source and produce a file of basic symbols. Note that space, whether in main store or in backing store, must be found for this intermediate form of the program. The next pass which will include at least the syntax analyser will then read this file and produce its own output file.

However, in some compilers all the phases can be gathered together into one pass, and instead of storing complete files of intermediate data, the phases call each other as subroutines to ask for or provide information one piece at a time.

Thus the syntax analyser may call the lexical analyser and ask it for the next basic symbol. It may also call the code generator to emit the next piece of code.

The organisation of phases into passes may depend on the language being compiled. Some languages actually *require* several passes. For instance, if an object in a program can be used before it is declared (e.g. a jump to a label may occur before the label's definition, or a procedure call may come before the procedure declaration) then code cannot possibly be generated for the use of the object without having complete knowledge of its properties. In such cases a complete pass will have to be made to gather such knowledge and another to generate code based on that knowledge.

In some languages one cannot even perform syntax analysis properly before a complete lexical pass has been made. For instance in a language where we could define new operators and priorities for them, one would not know whether to treat an expression such as "a << b + + c" as "(a << b) + + c" or as "a << (b + +c)", had the user been so foolish as to leave the declarations of the priorities of "<<" and "+ +" until later in the program.

We should note that the situation is sometimes confused by bringing the operating system into the picture. A multi-pass compiler may be organised as a number of cooperating processes which run at least conceptually in parallel. They would have to be carefully synchronised but a good system would do this automatically be making them communicate through **pipes** (UNIX† nomenclature, see [7]) which replace those intermediate storage files which are the main disadvantage of multipass compilers. However the gain will probably be more than offset by the system overhead necessary for scheduling the processes in and out of action.

1.3 RECURSIVE DESCENT COMPILING

In this book we are going to concentrate very heavily on one particular technique and on its application to a particular language, S-algol. We shall give a brief introduction and summary of its usage in the next section. Here we talk about the method which forms the subject of this book – **recursive descent.**

This method centres around the syntax analysis phase of the compiler which is divided up into a number of **recognition** routines, each of which has the task of checking whether a particular kind of phrase is present in the input. Each recognition procedure can call upon the services of other ones to recognise the appearance of subphrases and so on. For example we will see that an S-algol program consists of a **sequence** followed by a questionmark. The central recognition routine will therefore call the sequence recognising routine and check for the appearance of the questionmark on the input stream. The sequence recogniser will, in turn, call routines to check for declarations or clauses, because a sequence is basically a list of such entities. Most of these routines will be mutually recursive, reflecting the fact that within one sequence we can find others embedded at a

†UNIX is a trademark of Bell Laboratories

lower level. In the same way, expressions can contain subexpressions, declarations include inner declarations and so on. Each of these has its own recogniser which is invoked from above when appropriate.

Some recognisers will have choices to make. In the above example of the sequence recogniser, it will have to choose between calling the recogniser for a declaration or for a clause. When such choices are to be made, decisions are always taken by looking at the input stream for the next basic symbol. We shall see that declarations in S-algol always start with one of the reserved words 'let', 'procedure', 'structure', 'forward' or 'external', and that no clauses start with any of these symbols. Hence the sequence recogniser can choose the declaration recogniser if it finds one of these, or the clause recogniser if it does not.

The task of a compiler is not however merely to *recognise* correct programs; it must also produce object code. Therefore, each recogniser will be modified or refined in order to emit code. One can notice here that the syntax tree referred to as the output of the syntax analysis phase in section 1.2 is never explicitly grown. This is because the syntax analysis phase and the code generation phase are not separated into distinct passes, but rather integrated into one another in order to understand clearly what each recogniser-emitter does. The tree is implicit in the dynamic calling structure of the recognition routines and is traversed by the code generation phase as it is built, and branches no longer of use are destroyed as the routines are exited.

The addition of code generation to the recognition routines represents a **refinement** of them. Other refinements will be introduced and these are based on error recovery and type checking. If a recogniser finds some program constructs that it doesn't expect, what should it do? Should it merely print the message: 'You have made a serious mistake.' as one early compiler was reputed to do? Or should it offer 'IEH377I' or some such terse comment to the user? Are there alternatives to these and can the compiler recover from errors?

Each recogniser must check that expressions, clauses, declarations and so on have sensible type structures. One must not, for instance, add a string to an integer if that is not allowed in the language. The type handling part of a recogniser must also be able to pass type information back to its parent recogniser.

One of the main features of the recursive descent method when used practically is that it must be able to do its recognition, type checking and code emission without 'backup'; that is, if a recognition routine A decides to call another, B, it can be sure *from the first* that, barring errors on the user's part, it has made the correct choice based on the input it has before it. This limits the kind of language which can be compiled by the method, but not too severely. We will devote the early chapters to seeing just what kind of restrictions are placed on languages by this requirement, and how to get round them. Such restrictions are called the LL(1) conditions. We will explain this term in section 4.2.

Our particular compiler will also have the property that it is one-pass in