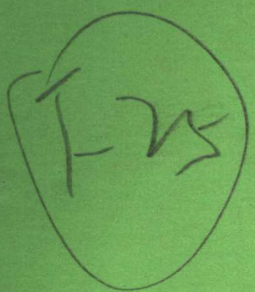
PROGRAMMING LANGUAGE TRANSLATION

R. E. BERRY, B.Sc., M.Sc.



PROGRAMMING LANGUAGE TRANSLATION

R. E. BERRY, B.Sc., M.Sc.
Department of Computer Studies
University of Lancaster

ELLIS HORWOOD LIMITED
Publishers · Chichester

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

First published in 1982 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
Berry, R. E.
Programming language translation. –
(Ellis Horwood series in computers and their applications)
1. Compiling (Electronic computers)
I. Title
001.64'25 QA76.6

Library of Congress Card No. 81-13469 AACR2

ISBN 0-85312-379-9 (Ellis Horwood Limited – Library Edn.)
ISBN 0-85312-430-2 (Ellis Horwood Limited – Student Edn.)
ISBN 0-470-27305-4 (Halsted Press)

Typeset in Press Roman by Ellis Horwood Ltd.
Printed in Great Britain by R. J. Acford, Chichester.

Table of Contents

PART 2

Author's Preface

This book provides an introduction to some of the more important techniques used in the construction of program translators. The book is directed at the increasing number of computer users who are curious to know more about the compilers or assemblers which process their programs. This has led me to write an overview, rather than an exhaustive study, of particular topics. As a source of examples I have made considerable use of Pascal S. Most if not all Pascal compilers are written in Pascal. While this property of being written in the language they compile is not exclusive to Pascal compilers, it makes them invaluable in teaching. I have used Pascal S in teaching for a number of years in different environments and its value as a teaching vehicle has become apparent through the interest it arouses in students. A listing of the Pascal S compiler/interpreter is provided, and I am grateful to Professor N. Wirth for permission to include this.

My thanks are due to my wife, and to the colleagues and students who have, directly or indirectly, helped in the preparation of this material. The responsibility for it is, however, mine alone. The series editor Brian Meek also deserves special thanks for his careful, pertinent, and helpful comments on my early manuscript.

Introduction

A program translator accepts as input a program written by a user and produces as output a version of that program which can be directly or indirectly executed by a computer. This translation process will include verifying that the user's program does not violate the syntax rules of the language in which it is written, checking that all symbols defined by the user are used in a consistent way, and generating a version of the program suitable for execution or interpretation. The user expects the translation process to produce a program which, when executed, will have the same effect as his original program were it directly executable. The translation from one program form to another is a complex task. Too often it is described by enumerating the techniques necessary to accomplish it with little attempt to identify the inter-relationships of these techniques. In an attempt to avoid this pitfall I shall discuss various aspects of the translation process and identify their implementation in a specific translator, the Pascal S compiler. To lend emphasis to this approach I have divided the book into two parts. Part 1 considers translation in general, but both text and examples progressively assume familiarity with the compiler given in part two. Part 2 gives a listing and documentation of the Pascal S compiler. Both parts assume familiarity with Pascal.

Pascal S is a subset of Pascal. In consequence the Pascal S compiler is small in comparison with many of the Pascal compilers currently in use. Nonetheless, significant effort is required to gain an insight into the compiler's action by studying its listing and such documentary help as is given here. This effort is well rewarded if it helps the reader understand a complex and important piece of software.

PART 1

Chapter 1

Lexical Analysis

1.1 USER INTERFACE

In considering the work of a compiler or an assembler it is too easy to assume that all such items behave in the same manner as those with which the reader is familiar. A user could communicate with these items of system software in several ways. Cards, paper tape, and interactive terminals may all be used to prepare a program which is subsequently submitted for processing. Further, cards may be encoded in different ways, paper tape may appear in different sizes (8 track, 7 track and currently the less common 5 track) as well as with different encodings, whilst terminals may transmit a line at a time or a character at a time for processing. Whatever the media selected, the user is simply trying to present his program for processing and can legitimately expect that the form of input media he chooses should not affect the integrity of his program. Equally, much of a compiler's effort is directed to determining whether the program submitted for processing is a valid program according to the rules of the programming language used. This judgement must be made whatever input medium was used. It will, I hope, be clear to the reader that accepting one character at a time from an interactive terminal will mean recognising *rub-outs* or perhaps *backspaces* as deleting the preceding character, whereas accepting a line at a time should mean that no such deletion characters are ever transmitted.

Some of the problems in dealing with a wide range of input media are dealt with in Hopgood and Bell [5].

1.2 ALPHABET

From this point it will be assumed that we have access to a character stream which is free of the idiosyncrasies of any data preparation device. But before proceeding further we need to establish what characters will be considered as legal input for processing. For example a Fortran compiler must flag the character '[' as illegal, since no use is made of it in Fortran. Is the compiler to accept upper and lower case letters? Should a Pascal compiler accept the character '{'

as well as the character pair '(*' to denote the opening of a comment? These questions must be resolved in order that all characters which do not form part of the legitimate input to a compiler can be identified and flagged as erroneous. Having identified the character set or alphabet we are prepared to deal with, it is then legitimate to ask what we are to do with these characters.

In lexical analysis we consider the user's source program simply as a character stream. We must scan this input stream in order to find groups of characters which may be called *textual elements*. These are words, punctuation, single and multi-character operators, comments, spaces etc. which comprise the user's program. In its simplest form the scanner (lexical analyser) finds these groups of characters and classifies each according to its textual element, and passes on a much reduced form of the original program for subsequent processing. In particular, comments which are recognised are eliminated. Spaces, line feeds, carriage returns and other editing symbols once recognised have no further value and are also eliminated. It is important to note however that spaces may play an important role, since in some languages, notably PL/1 and Cobol, they have the role of delimiters. That is, they mark the end of such textual units as reserved words. Most if not all implementations of Pascal adopt this approach. In other language implementations, reserved words may be preceded and followed by a special character other than space.

For example begin appears as 'begin' in many implementations of Algol 60 and Algol 68. Whatever delimiters are used, it will be the function of the scanner to collect the characters between consecutive delimiters, of which the space character is only one, and to determine whether this group of characters is a textual unit, and if so what kind. Once recognised as a valid textual unit each group of characters is replaced by a token or internal symbol which is used for easier recognition in subsequent processing. The precise form this token takes is a matter for a particular implementor, but it can take the form of a reference to a particular table together with a position number, or a special bit pattern in a byte or a word, or, as in the case of Pascal S, it can be given the value of an element in a set of predefined symbols. In order to help clarify the role of the token, consider the programming language statement:

```
if x>y then x:=x-y else y:=y-x
```

The language presents the statement in this form because it is readable and easily intelligible to users. The same statement could equally be written in the form:

```
|x>y~x\x-y@y\y-x
```

which is perhaps not quite so easily understood. All that has happened is that the compound symbols of the first example (i.e. if, then, else, :=) which consist of more than one character, have been replaced by single character equivalents in the second example. In short, the reserved words have been replaced by tokens. For this example tokens are visual, but for our scanner they clearly need not be.

The recognition of such textual units and their replacement by tokens implies that we can recognise any one of the possible reserved words. This will mean that the scanner must have access to a list of all reserved words and can correctly identify a group of characters as being one item from this list. This involves a *table look-up* which may be undertaken in several different ways. For the moment we shall take it that the look-up can be done and postpone a discussion of how until later. Hence we can assume that we have reduced each reserved word to a simple token.

One item which can be dealt with in a straightforward manner for many high level languages is the string. If we consider the Pascal statement:

```
write ('this is a string')
```

then the collection of items between apostrophes is a string. It is usually of no importance what characters or how many characters are in the string since in most languages such strings cannot be manipulated. Accordingly a scanner, having successfully recognised such a string, is likely to store away the string in, say, a string table, and leave a token in the program source which will denote the presence of a string and where it may be located.

The textual units which occur most frequently in the user's source program will be identifiers, or user-defined symbols. Such identifiers will range from one character to some implementation- or language-defined maximum number of characters in length. Since in the remainder of the compiler it will be time consuming to be constantly dealing with multi-character identifiers, it is useful to cause the scanner to construct a table of all the identifiers which are recognised, and to replace them in the user's source program by a token, together with sufficient information to locate the appropriate symbol in the identifier table. Note that, for the reserved words of the language, the scanner attempted to look up the appropriate group of characters in a previously constructed table, while in the case of identifiers a slightly different operation is required. The identifier table (or that part of it to which we currently have access) must first be scanned to see if an entry already exists for this group of characters. If it does, well and good; if it does not then an entry must be created.

Apart from symbols such as ':=', '..', which, although consisting of more than one character, are easily dealt with because they are few in number and can be treated as special cases, the only multi-character textual units which have not been dealt with are numeric constants.

Numeric constants divide into two types, those with fractional parts and those without. It is the scanner's function to recognise valid representations of each within the source program being processed. This can often be done by making use of a *state table*. It will be instructive to consider an example of this technique but in order to do so at this point we need to anticipate material which will be covered in more detail later. In particular it is assumed that the reader is familiar with definitions given in the following form:

⟨ unsigned number ⟩ ::= digit ⟨ rest unsigned number ⟩
⟨ unsigned number ⟩ ::= . ⟨ decimal fraction ⟩
⟨ unsigned number ⟩ ::= E ⟨ exponent part ⟩
⟨ rest unsigned number ⟩ ::= digit ⟨ rest unsigned number ⟩
⟨ rest unsigned number ⟩ ::= . ⟨ decimal fraction ⟩
⟨ rest unsigned number ⟩ ::= E ⟨ exponent part ⟩
⟨ rest unsigned number ⟩ ::= Λ
⟨ decimal fraction ⟩ ::= digit ⟨ rest decimal fraction ⟩
⟨ rest decimal fraction ⟩ ::= Λ
⟨ rest decimal fraction ⟩ ::= E ⟨exponent part ⟩
⟨ rest decimal fraction ⟩ ::= digit ⟨ rest decimal fraction ⟩
⟨ exponent part ⟩ ::= sign ⟨ exponent integer ⟩
⟨ exponent part ⟩ ::= digit ⟨ rest exponent integer ⟩
⟨ exponent enteger ⟩ ::= digit ⟨ rest exponent integer ⟩
⟨ rest exponent integer ⟩ ::= digit ⟨ rest exponent integer ⟩
⟨ rest exponent integer ⟩ ::= Λ

This is not the most compact way of expressing these definitions but in this form their use in constructing a table is most easily seen. Notice that there are seven different items enclosed in angle brackets, and that each of these appears at least once on the left hand side of the definitions listed. These items:

1 unsigned number
2 rest unsigned number
3 decimal fraction
4 rest decimal fraction
5 exponent part
6 exponent integer
7 rest exponent integer

determine the number of states, or rows in the table, while the number of columns is determined by the number of terminal symbols or characters we need to use in the definitions listed. Note that nothing is lost by regarding ⟨ sign ⟩ and

⟨digit⟩ as terminal symbols. The symbol _Δ_ is used to represent any other character. With this information the table which may be used by the scanner is now constructed.

	Sign	digit	·	E	_Δ_
1	/	2	3	5	/
2	/	2	3	5	exit
3	/	4	/	/	/
4	/	4	/	5	exit
5	6	7	/	/	/
6	/	7	/	/	/
7	/	7	/	/	exit

The first row is the entry for ⟨unsigned number⟩, and in the list of defining rules this item appears three times on the left hand side of a definition. The corresponding right hand sides require that 'digit', '·', 'E' are recognised and accordingly there are entries in the first row of the table for the clements with these characters at the head of the column. In each case the entry is the row number of the item in angle brackets which follows the character in the appropriate definition. Thus in the first row under E the entry is 5 which refers to the fifth row and thus the item in angle brackets with reference 5, i.e. exponent part.

Use of the table is straightforward, since all that is required is that the current state (row number) and current character from the input stream be available. This information is used to determine an element position in the table which will be the new state or row number. Starting at state 1 it can be seen that only those characters which can be legitimately expected at that point in the processing will produce a satisfactory change of state. Any other character will take us to a table entry containing / which can be regarded as an error exit. As long as the definition of floating point constant in the language being processed can be redefined in the form of a set of rules such as those given earlier, this technique may be used. This is efficient and can produce good diagnostics though it does need table space.

Let us take 3.141* as the character string to be parsed. The start state is 1, and the first character is a digit. Inspection of the first row of the state table shows that the new state is to be 2. In this state we consider the next character of the input string which is the decimal point. The entry under the decimal point in the second row of the state table is 3, which is the new state. This process can be more concisely represented by the following table:

current state	current character	new state
1	3	2
2	.	3
3	1	4
4	4	4
4	1	4
4	*	exit

The sole purpose of the finite state table is to help the scanner recognise a particular item. The result of using the table in the example as described would be to produce a simple *yes* or *no* answer to the question *was the number processed a valid floating point number?*. A scanner will frequently combine recognition of such an item with its evaluation and conversion to an *internal form*. However, if this additional work is undertaken then one must make assumptions about the computer on which the compiler is to run. This is because the way in which a floating point number is stored will vary from machine to machine. As a result there are persuasive arguments for postponing such conversion operations until later in the compilation process so that as much of the compiler as possible can be made independent of particular machines. As far as our scanner is concerned it will be sufficient to assume that successful recognition of a floating point number will cause its replacement by a suitable token and table reference. It should be clear that integer constants can be dealt with by the table constructed, though they too will be unsigned constants.

The role of the scanner can be regarded as that of accepting the character stream as input, which is the user's source program, and producing as output a string much reduced in length and consisting of a series of tokens, together with one or more tables. However, this simple view comes from looking at the scanner in isolation. In practice, the scanner is more likely to be invoked as and when required by other elements of the compilation process. This can be clearly seen by considering the roles of nextch and insymbol in the Pascal S compiler.

EXERCISES

1. Identify, and list the differences between, the floating point numbers accepted by the finite state recogniser in the text, and the floating point numbers defined in the Pascal S syntax diagrams.

2. Describe which tokens, if any, the Pascal S compiler uses for each of the following symbols ':=', '<>', '..', '(*', '<='.

3. Operands of an expression may be single digits, or single characters. The permitted operators are +, −, *, /,. Construct a finite state recogniser for expressions which do not contain a unary operator, and also one for expressions which may contain a unary operator.

4. Is a comment within a comment legal in Pascal S? What is the outcome if a program with such a construction is processed by the compiler?

5. List the modifications necessary to the Pascal S compiler in order that both upper and lower case characters may be used for input. The modifications should ensure that, for example, BeGIN will be correctly recognised.

Chapter 2

Syntax Definition and Syntax Analysis

In writing a computer program a user must use a programming language. Such languages have only been available since about 1950. They have been designed by man for his own use. Our natural language has evolved over many hundreds of years and, comparing these timescales, it should come as no surprise that programming languages are not yet so rich nor perhaps as flexible as we would like. Nonetheless, programming languages have reached the stage where much use is made of them, and features which are hard to implement or hard to use now rarely pass the design stage of the language.

When designing a language, serious consideration must be given to the character set to be used, the ways in which these characters can be combined to make symbols, and the rules which govern how these symbols may be combined to produce the *sentences* of the language. There are a variety of ways in which a compiler may verify that a program is syntactically correct; that is, correctly constructed according to the rules of the language being used. It will be instructive to examine how the rules which govern the syntax of a language may be expressed. Consider the following rules:

1. ⟨number⟩ ::= ⟨num⟩
2. ⟨num⟩ ::= ⟨num⟩⟨digit⟩ | ⟨digit⟩
3. ⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

note that rule 2 can be written more concisely as:

⟨num⟩ ::= ⟨digit⟩ {⟨digit⟩}

Rule 3 simply defines an item called ⟨digit⟩ which is one of the characters 0 to 9. Remember the purpose of the rules is to define rigorously and concisely how new symbols may be constructed. For this purpose no reliance can be placed on intuitive notions of what a digit, or for that matter a number, shall be. Rule 2 gives two alternatives for how a num can be formed. It may simply be a digit, or it may be a num (therefore a digit) followed by a digit. In short