SOME ASPECTS OF THE IMPLEMENTATION OF THE COMPILER COMPILER ON ATLAS

by

I.R. MacCallum.

Thesis presented in support of an application for the degree of Master of Science in the Victoria University of Manchester

Computing Machine Laboratory. January 1963.

CONTENTS

ACKNOWLEDGEMENT

I am indebted to Mr. R.A. Brooker and in particular to Dr. D. Morris, both of the Computing Machine Laboratory for suggesting the subject and for extensive assistance and helpful criticism in the preparation of this thesis.

I would also like to thank Miss M. Bruce for her careful typing of the manuscripts.

I am also indebted to the Department of Scientific and Industrial Research for a grant for the period in which the work for this thesis was done.

CHAPTER I

Introduction

1.1

The present day multiplicity of source languages for automatic digital computers and the continued need for special purpose languages presents the compiler writer with a problem of considerable magnitude. The purpose of this thesis is to describe recent work on the implementation and general development of one system which aims to reduce significantly the time taken to write such compilers. It also simplifies the corrections and amendments which often have to be made to a compiler once it is in regular use. This system is known as the COMPILER COMPILER.

Briefly, the system allows the syntax of statements in a phrase structure language (e.g. a scientific autocode) to be specified in terms of formats and phrase definitions, and the semantics in terms of format routines. Given such a specification of a compiler, the system will generate (in machine instructions) a compiler which will translate statements of the source language into machine instructions and immediately enter the compiled program. A source language may be extended by introducing into the program further phrase definitions, formats and format routines to define additional source statements which can be used subsequently in the program. Appendix 1 consists of a

detailed description of the way in which a compiler is to be defined for the system.

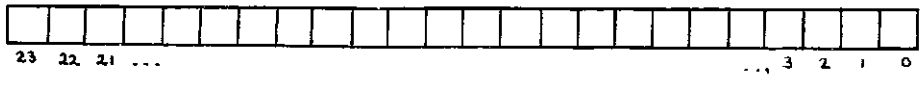
Although the compiler compiler has been written in the first place for the Ferranti Atlas computer, it was realised at the outset that fundamental concepts such as list structures, syntactical analysis, and manipulation of integers and binary numbers are in no way Atlas oriented, and consequently it has been coded in such a way that subsequent adaptation to another suitable computer can be effected with a minimum of work. This has been done by coding one section of the compiler compiler in its own language, (namely, the language in which a compiler is defined,) and the other section in a language which is largely a subset of the class of built-in instructions (see Appendix 1, p. 10), and by using a Mercury computer to translate from this language into the required machine code. The division of material into these sections is explained in chapter 2.3 and in chapter 3 the Mercury translation program is described. It was hoped that it would be possible to make use of this machine independence in the development of the compiler compiler for three reasons. First, it was desirable for the compiler compiler to work on Atlas as soon as possible so that the Manchester University Mercury Autocode Computing Service could be transferred to Atlas at the earliest possible date; second, the writing of the compiler compiler was almost complete some seven

months before the main core store was available on Atlas; third, there was a possibility of using time on an IBM 7090 computer (with a 32K store) for such development. It was therefore decided to adapt the system for the IBM computer mainly for the purpose of development until sufficient store was installed in Atlas, but also for the knowledge of whether such a project would be practical. Chapter 4 contains an account of the proposed version for the IBM 7090 which was almost complete when it was learned that the computer would no longer be available. At this time a small store of 1024 words was being commissioned on the Manchester University Atlas (MUSE), and in order to make use of the next few months, a scheme for testing selected parts of the system on this store was devised, as described in chapter 5. This small subsidiary store is used by programs in a slightly different manner to the main store, and a program written for the latter will generally require modification if it is to operate on the former. The necessary modifications were few, and relatively simple to implement, largely due to the machine independence of the coding and the flexibility of the Mercury translation program.

1.2

A full description of the Atlas computer is to be found in the Ferranti Manual (ref.1) but it is worth noting here some features of the computer which have influenced the

coding of the compiler compiler and its development on Atlas. Although it has a core store of 16,384 words, the hardware and drum supervisor program (ref.2) enable the drum backing store and the core store to be regarded by the programmer as a store of over 114,000 words in the MUSE installation. The standard word lengths are 24 and 48 bits. Instructions and floating-point numbers occupy full 48-bit words, while integers and addresses occupy 24-bit half-words. The structure imposed on half-words is shown in fig.1 from which it can be seen that an address s, lies in the range $0 \leqslant s \leqslant 2^{21} - 1$, and an integer n lies in the range $-2^{21} \leqslant n \leqslant 2^{21} - 1$.



digit 23 — (Integer : sign digit
(address : always 0

digits 2-22 — integer or address
digits 0-1 — spare

fig. 1

Atlas has 128 B-modifier registers, some of which are reserved for special purposes (ref. 1 section 2.2.2) and those generally used by the compiler lie in the range B0-B99 (B0 is identically zero). Since all input and output operations are time-shared simple instructions cannot be provided for the user. In their place, a pseudo-instruction, known as an extracode, must be used which calls a fixed

store sequence to perform the necessary operations. Extracodes are also used by the compiler compiler for other common operations such as shifting, subroutine entry, integer multiplication and division etc.

1.3

Before it is possible to decide whether the compiler compiler is suitable for use with a certain computer, a number of important features of the system must be taken into account. The most significant of these is the size of the compiler compiler. The method adopted for loading it into the computer (described in chapter 2) renders it difficult to calculate exactly how much space is required by format routines and phrase definitions. Its size also depends to some extent on the efficiency of translation desired, which is discussed in chapter 6. However it is estimated that a minimum of about 6,000 instructions and 6,500 half-words are needed by the Atlas version for the compiler compiler itself, and for the Mercury Autocode compiler a further 4,000 instructions and 8,000 half-words are required, making a total in the region of 10,000 instructions and 14,500 half-words. An estimate of the space required in the machine under consideration can be made by comparing the coding of the same instructions for Atlas and the other computer. Fig.2 shows the Atlas and IBM 7090 machine instructions for one of the built-in phrase routines of the system which is fairly typical of

```
          TOCD    *
          CLA     B02
          STO     B96
          CLA     B03
          STO     B93
          CLA     =0
          STO     B94
          CLA     =0
          STO     B95
1)I72     LAC     B02,1
          CLA     1,1
          STO     B91
          CLA*    B91
          STO     B92
          CLA     B92
          SUB     =15
          TZE     2)I72-)I72,4
          CLA     B92
          SUB     =8
          TZE     2)I72-)I72,4
          CLA     B92
          SUB     =32
          TZE     2)I72-)I72,4
          CLA     B92
          CAS     =25
          TRA     3)I72-)I72,4
          NOP
          CLA     B92
          CAS     =15
          TRA     4)I72-)I72,4
          NOP
3)I72     CLA     =1
          STO     B94
4)I72     CLA     B91
          STO     B02
```

```
          0121,96,02,0

          0121,93,03,0

          0121,94,0,0

          0121,95,0,0

(  1)0101,91,02,1

          0101,92,91,0

          0172,92,0,15
          0224,127,127,L2

          0172,92,0,8
          0224,127,127,L2

          0172,92,0,82
          0224,127,127,L2

          0170,92,0,25
          0227,127,127,L3

          0173,92,0,15
          0227,127,127,L4

(  3)0121,94,0,1

(  4)0121,02,91,0
```

```
          CLA     B02
          STO*    B03
          CLA     B95
          ADD     =1
          STO     B95
          TRA     1)I72-)I72,4
2)I72     CLA     B03
          SUB     B93
          TZE     5)I72-)I72,4
          CLA     B94
          SUB     =0
          TZE     6)I72-)I72,4
          CAL     B95
          ANA     TAGGOFF
          ORA     TAG10
          SLW     B97
          CLA     B97
          STO*    B96
          CLA     B03
          ADD     =1
          STO     B03
          STZ     B10
          TRA*    B70
5)I72     SSM
          STO     B10
          TRA*    B70
6)I72     CLA     B90
          STO     B02
          CLA     B93
          STO     B03
          CLA     ORIGIN+149
          STO     B97
          LAC     B97,4
          TRA     2,4
```

```
          0113,92,03,0

          0121,95,95,1

          0124,127,0,L1
(  2)0172,03,93,0
          0224,127,127,L5

          0172,94,0,0
          0224,127,127,L6

          0165,97,95,4194303
          0167,97,0,0.10

          0113,97,93,0

          0121,03,03,1

          0170,0,0,1
          0121,127,70,0
(  5)0172,0,0,1

          0121,127,70,0
(  6)0121,02,90,0

          0121,03,93,0

          0101,97,0,1048725

          0121,127,97,2
```

Fig 2. Comparison between IBM 7090 and ATLAS machine codes for routine I72 .

the kind of instructions which appear in the compiler compiler. The ratio of Atlas to IBM machine instructions is about 1 : 1.7, and since it is not possible to refer explicitly to half-words in the latter machine the total number of words required is of the order of 31,000, leaving about 1,000 words for object program space. For development purposes, the IBM 7090 would have been adequate, but before the compiler compiler could be used as a practical means of program translation, some modifications for dumping object program on magnetic tape would have to be incorporated. The facility for introducing new phrase definitions and formats into a source program may be dispensed with making a further 3,000 words available for object program in the IBM computer.

The two spare bits referred to in fig. 1 are used in the Atlas version as tags (ref. 3, p.224; ref. 4, pp.33, 43). No assumptions have been made in the coding of the compiler compiler as to their position in a word except that they do not interfere with its use as an address. However, before any arithmetic operations are performed on two words which may be tagged, the tag will be removed from at least one of the words. There must, then, on the computer under consideration, be two spare digits in a word, suitable for use as tags.

The method of using the store has been described elsewhere (ref. 3, p.220) where it is stated that all items in the record store are "store invariant". Since all

hand-coded items are initially in this part of the store, all instructions which cause control transfers or set links must have dynamic access to the address of the current instruction, or at least to some fixed part of the routine, such as the first instruction. In Atlas it is possible to make relative control transfers using just one instruction; for example, 121, 127, 127, 10 causes control to be advanced by 5. In order to perform the same operation with the IBM 7090 three instructions are necessary, namely,

```
STL   B99
LAC   B99, 1
TRA   7,1
```

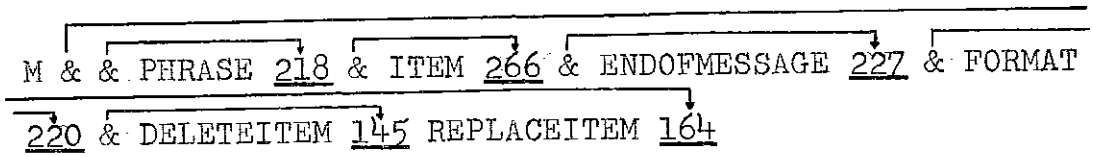
Since control transfers are frequently occurring operations, space may be saved by utilising the B-register which contains the address of the current routine. (see chapter 4.)

1.4

It may be regarded as a natural consequence of writing an input program that at some stage it becomes possible to read in the rest of the program using that part already within the computer. This has certainly been true in the case of the compiler compiler and therefore some understanding of the system is essential for a full appreciation of the implementation and development. This section provides a brief outline of the compiler compiler.

A compiler for a phrase structure language is described essentially by three types of primary statement, namely phrase definitions (or simply, phrases), formats and format

routines. Each type of statement consists of a heading or master phrase followed by the statement proper, the end of which is denoted by the next master phrase. This primary material is read a line at a time by a line reconstruction process which forms a chain list in which one cell represents each printed character or space on the line. This list is then submitted to an editing routine which generates a new list removing spaces and erazes, and converting symbols with meta-syntactical significance into their internal form. The edited list is now analysed by the expression recognition routine (ERR)[*] with respect to the format class of master phrases. This format class initially has the form



The ERR plants the analysis record (a single word, being the serial number of the routine to interpret the 'meaning' of the statement) which enables the appropriate routine to be entered. Thus it is a simple matter to enter a routine of a supervisory or diagnostic nature by means of the appropriate master phrase, e.g. END OF MESSAGE. Additional

---

[*] see ref. 4, p.36 ff.

master phrases may be added, as in the case of the format routine heading, by means of the format,

FORMAT [MP] = ROUTINE, 221

A source program is translated into machine instructions by a similar process to that used for processing the primary material. Since the punching convention for input material of source programs will, in general, differ from that used in the primary material, a different editing routine is used. The edited line is analysed with respect to the format class of source statements by a special recognition routine. It is desirable that those parts of the compiler compiler which are used for the translation of source programs should be as efficient as possible and so the source statement recognition routine has been 'fixed' in the store in order to achieve more effective use of the machine instruction code than if it were 'store invariant'. The first word of the analysis record planted by this routine is the serial number of the format routine for translating the source statement just recognised, and the remainder of the analysis record provides the format routine with its input parameters.

CHAPTER 2

The Bootstrap.

2.1.

Before any program may be read into a computer it is essential that there should be some instructions already in the machine to interpret the program and establish in the store the required pattern of digits which the computer can obey. The question which immediately arises is, "How is the most elementary input routine read into the computer?" The answer to this is often by means of a procedure known as a 'bootstrap'. This term is used to describe a recursive process in which one program is used to read in material to compose a more complex program until an input routine suitable for more general use is in the store. The first input program must be introduced into the computer by some means other than the normal input channel (e.g. paper tape, cards) and is usually by means of fixed store program (Atlas), by manually operated instructions which read from hand keys to store (Mercury) or in the hardware (Titan). A good example of a classical 'bootstrap' is given by the starting procedure for the Mercury computer which furnishes it with a binary input program. It consists initially of a loop of 4 instructions which are read manually from the hand keys to the core store.

A further manual order is required to set control to the first of these instructions and the simple loop reads the bootstrap tape which consists of 3 distinct sections, each corresponding to a different phase of the recursive operation. Once the bootstrap tape has been read, the computer is able to accept binary tapes. In this example, an attempt has been made to reduce the number of manual instructions and the length of the bootstrap tape to a minimum.

An indication of the size of the compiler compiler has been given in the introduction, and this chapter describes how the principle of a bootstrap has been used to reduce coding errors and to simplify their correction. First, those principles which are computer-independent will be described, and then in some detail, the method by which the compiler compiler is bootstrapped into Atlas.

2.2.

The compiler compiler is 'complete' in the sense that it may be written in its own language. In other words, it consists of phrase definitions, format dictionaries, and routines whose instructions belong to an extended set of the built-in instructions and auxiliary statements. The general implication of this is that once the material for interpreting one of these primary statements has been loaded, it is possible to process subsequent statements of that type written in the system language. This type of

bootstrapping procedure has been adopted for the compiler compiler for three reasons.

First, it has been possible to write a considerable part of the system in a language which is particularly suited to its own requirements and which reduces the likelihood of errors in the coding. Second, it provides more positive evidence that the various parts of the system function properly; for example, if a phrase which has been processed by the phrase assembly routine is subsequently used by other parts of the program and is found to give the expected results then it is almost certain that the phrase has been assembled correctly. Third, once the compiler compiler is working on Atlas, it will be possible to produce a compiler for another computer by providing the Atlas version with a set of primary assembly routines (ref. 4, p.46) which will plant machine instructions for the other computer. By reading the compiler again, written entirely in the language of the system, a compiler for the other machine will be generated. It is quite likely, that if the compiler compiler is used on the Titan computer, its compilers will be written in this manner.

2.3.

Having decided that a bootstrap is both suitable and desirable for the compiler compiler, it is necessary to decide in what order the material should be assembled, in order to simplify the programming and optimise the resulting compiler.

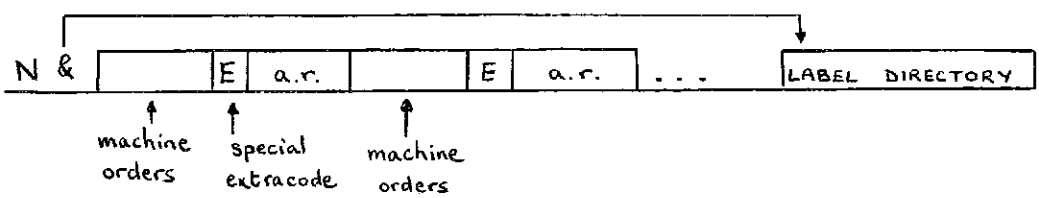
It should be apparent that the routine which assembles the routines of the system cannot itself be read into the computer in the language of the compiler compiler. This routine therefore, and its subroutines belong to the section which is to be hand coded. However, there are two ways of coding the remaining material which enables this routine to operate and phrases and formats to be assembled. One is to hand code all the built-in phrases and format dictionaries for the built-in instructions and then to read the routines for processing phrases and formats through the system. The other is to hand code the phrase and format assembly routines and use them to read the built-in phrases and formats required by the routine assembly routine. One disadvantage of the first method is that most of the subroutines required for the phrase and format assembly routines are also required by the routine assembly routine, and consequently the amount of material to be read in by the system would be quite small. However, the main reason for choosing the latter method was because routines are easier to code by hand and subsequently alter than dictionaries. Further, because of the similarity between the language of the hand coding and the built-in instructions, little would be gained at this stage, from coding these routines as format routines.

When a routine is being assembled, each instruction is examined to decide whether it is possible to plant machine instructions rather than the analysis record of the instruction which has later to be interpreted. For most of the built-in instructions therefore, there is an interpretive routine and a primary assembly routine (or compiling version of the routine) which provides a direct translation. It is unnecessary for both the interpretive and the compiling version of each routine to be hand coded, for the compiling versions may subsequently be interpreted or the interpretive routines may be compiled. In fact it is possible that if only three interpretive routines were hand coded, namely

[AB] = [WORD]
[AB] = [WORD] [OPERATOR] [WORD]
→ [LABEL] [IU] [WORD] [COMPARATOR] [WORD]

then the remaining interpretive routines and all the compiling routines could be bootstrapped in by these.

At this point it is important to realise that maximum efficiency must be achieved at that time when object programs are being translated rather than the time when a compiler is being assembled. It is therefore necessary to examine the structure of a format routine which would be used to translate a source statement into machine code.



The machine orders are those which are planted by the compiling routines and therefore the orders planted, rather than the routines which plant them, must be as efficient as possible. The analysis records are 'obeyed' by the interpretive routines and therefore their efficiency is important. For these reasons it was thought best that all the interpretive routines for the built-in instructions should be hand coded and that the compiling routines should be written in the language of the system.

2.4

The compiler compiler may now be considered as being in two sections, namely that which is written in its own language, and that which is hand coded for translation on Mercury into a suitable form of input for Atlas. The latter part occupies some 3,750 instructions in Atlas and this section describes the way the system has been extended to simplify its development on Atlas.

The approach which has been adopted was influenced mainly by two factors. The first of these is the existence, within the compiler compiler, of a routine for recovering the space occupied by a redundant item in the record store (ref. 4, p.33). If there were some means of reading a new item to the head of the record store and updating the index, this would provide an easy way of replacing faulty items in the record store. A simple input routine incorporated into the compiler compiler, for use before the routine

assembly routine is operational, could then be used for this purpose. The other factor was the nature of the simple input routines which had been written for Atlas at the time when limited testing of the system began. Two forms of input were available for the 1024 word store in March 1962. The more elementary of these, Octal Input, is held in 64 words of the fixed store and makes the whole of the subsidiary store available to the user. An instruction, in this form of input, is represented by 16 octal digits which makes errors both hard to find and in some cases, where an instruction has to be inserted, awkward to correct. The other was a simple form of Atlas Intermediate Input which only left about 300 words of the subsidiary store for use. Neither the elementary form of Intermediate Input nor any of its proposed forms had any facilities for evaluating the relative distances between two labelled instructions in a program, and so this form of input was rejected.

As suggested already, an elementary input routine was embedded into the system itself, its rôle being that of a simple routine assembly routine which is entered upon recognition of a master phrase. For this purpose the master phrase ITEM has been introduced and the corresponding routine is known as the item routine. This routine, together

with those routines which are essential for its operation are read into Atlas by Octal Input and the remaining routines of the hand coded section are read by the item routine in a language resembling Intermediate Input but tailored to meet the requirements of the compiler compiler.

The symbolic language which is used, differs from Intermediate Input mainly in two respects. Since all instructions causing control transfers within a routine are of the form

function digits , 127, 127, address ,

reference to a label in the address part of an instruction is interpreted as referring to the number which must be added to, or subtracted from control in order to effect the jump. The other difference is concerned with the position of the implied binary point. In accordance with fig. 1, it is taken to be two binary places from the right whereas Intermediate Input assumes it to be three places from the right.

The language is best described by means of phrases. Statements may belong to three classes.

(i) H = [ADDRESS PART]

(ii) [FD] [COMMA] [N] [COMMA] [N] [COMMA] [ADDRESS PART]

(iii) I [N] = CC

where, [ADDRESS PART] = [-?] [N].[0-3], [-?]. [0-3], [-?] [N],
L [N]. [0-3], L [N]

[FD] = [BD] [OD] [OD] [OD]

[N] = 0, 1, 2, 3, ...

[BD] = 0, 1

[OD] = 0, 1, 2, 3, 4, 5, 6, 7

[0-3] = 00, 01, 10, 11, 0, 1, 2, 3

and COMMA denotes a, .

Any instruction may be preceded by a label which is written ([N]) and [N]$\leq$40. An asterisk punched in front of an instruction causes an entry to the post-mortem routine to be made if the instruction it precedes is about to be obeyed. An example of a routine written in this language has been given in fig. 2. An instruction of the type (iii) above is used to denote an alternative entry to a routine. During input, it causes the address of the next available register in the record store to be loaded into the[N]th position in the index.

The usefulness of having a routine such as this embedded in the compiler compiler can be summarized as follows:

(i) The quantity of material to be read by Octal Input amounts to only about one third of the hand coded section.

(ii) Alterations to program are easily made in this language and a facility for replacing an item is readily incorporated into the system.

(iii) It was a suitable routine for early development on the subsidiary store and

enabled certain other routines to be tested at that time.

(iv) During the development of the compiler compiler it was necessary to use interrupt control (B125) but main control (B127) was used throughout the symbolic coding and a few orders in the item routine substituted 127 for 125 whenever it appeared as the number of a B-line. When main control became available it was necessary only to remove these orders.

2.5

It is now possible to summarize the various stages of the input of the compiler compiler.

	Items input.	What may be done at each stage.
1.	Item routine and those items necessary for its operation (in octal).	Read items in symbolic language.
2.	Routines for deleting and replacing items. Miscellaneous subroutines of items in 1. which are not required by item routine.	Replace any item in the record store ~~(except the routine for replacing items)~~ and delete any redundant item.
3.	Phrase and format assembly routine and subroutines.	Read phrases and formats in language of system.
4.	Built-in phrases and formats; routine assembly routine and subroutines.	Read format routines
5.	Interpretive routines and transplant sequence (ref.4, p.42).	Format routines can be obeyed.

6.	Auxiliary formats; auxiliary format routines required by compiling routines.	Read and obey compiling versions.
7.	Compiling routines, built-in phrase assembly routine.	Built-in phrases can be read in the system language and obeyed.
8.	Remaining auxiliary format routines, compiling routines for second time.	Compiler can now be read.

Notes:

1. In practice, the built-in phrases and formats are not input until after the interpretive routines and the transplanting sequence, since it is convenient to have them on 7-hole tape and all the material in sections 1-5 is on 5-hole tape.

2. For a specification of the built-in phrases of section 7, see Appendix 1, p.37 f.

CHAPTER 3

The language of the hand coding.

3.1

In the Introduction it was stated that the fundamental concepts of the compiler compiler are machine-independent and that it is desirable to reflect this in the way in which the system is coded. It has also been noted that the program is of considerable size and that the language used should simplify the coding and reduce the occurrence of errors. As explained in the previous chapter, part of the compiler compiler has been written in its own language which is fundamentally machine-independent and also simple to program, and the remainder has been written in what has been referred to as the 'hand coded' language. In order to facilitate translation from the hand coded language into basic machine orders for Atlas (or another suitable computer), a Mercury Autocode program has been written. This chapter describes the hand coded language and the translation program.

The Mercury program resembles the compiler compiler in many respects. The syntax and semantics of the instructions of the hand coded language are specified in the first phase of its operation, and in the translation phase a crude form of syntactical analysis identifies instructions one at a time, leading to the corresponding machine orders being printed. In the analysis and in the syntax of the language all integers (except certain digits specifying tags) are

represented by the character n (or ').

Before entering into details, it should be noted that the process described in this chapter is intended for the initial development of the compiler compiler. At the time when it was hoped to develop the system on an IBM 7090 prior to Atlas, this method of producing both IBM mnemonic code and machine orders for Atlas proved to be most useful. If it were now required to adapt the compiler compiler for another computer a method similar to that mentioned in 2.2, using Atlas, would be employed.

3.2

The types of operation most frequently occurring in the compiler compiler are those involving store transfers, arithmetic and logical operations on integers and binary words, and conditional control transfer instructions. The language of the hand coding therefore, is primarily intended for such operations but the primary phase of the translation program enables extensions to the language to be easily made, as and when they are required. Two distinct stores are referred to, namely, the main store and a small store of 128 registers referred to as B0, B1, B2, ..., B127 , which corresponds to the B store of Atlas. B100-B127 inclusive are reserved for special use, and do not generally appear in the hand coding.

In order that the Mercury translation program should be as simple as possible, the syntax of the hand coded

language is only one level deep. Every instruction used should conform exactly to one of those specified in the primary phase of the program. Floating addresses are used for control transfers, and any instruction may be preceded by a label thus

n) instruction

A complete list of the instructions used in the hand coded language is given in appendix 2. It is convenient for explaining the language, to consider them in the following classes.

(i) Arithmetic and store transfer instructions.

These are largely a subset of the four built-in instructions

[AB] = [WORD]
[AB] = [WORD] [OPERATOR] [WORD]
([ADDR]) = [WORD]
([ADDR]) = [WORD] [OPERATOR] [WORD]

Parentheses are used, as in the built-in instructions, to denote the contents of the store address specified but the local variables α_1 ,α_2, ... cannot be used.

A further set of instructions in this class permits access to the Index*. It includes

INDEX[B?] n = [B?] n
Bn = INDEX [B?] n

(where [B] = B)

---

* Ref. 4, p.33

(ii) Logical operations.

Each of the boolean operators AND, OR, NOT EQUIVALENCE (EXCLUSIVE OR) may be used with B-registers or integers as operands.

e.g. Bn = Bn AND n
Bn = Bn NOT EQV Bn

A group of instructions for manipulating tags includes,

Bn = Bn WITH TAG [0 - 3]
and Bn = Bn LESS TAG

(where [0-3] = 00, 01, 10, 11)

(iii) Control transfers.

These are generally written in the form

→ n

where n denotes the floating address specified by label n in the same routine. Where the absolute address for the transfer is available in a B-line, the instruction

C = Bn

may be used. Routines, which are preceded by two information words are entered at the first machine instruction by

ENTER Bn

and phrase routines are entered from the ERR by

ENTER PR Bn

which also sets the necessary link.

(iv) Conditional control transfers.

The meanings of most of these are clearly conveyed by the format of the instructions, for example

→ n IF Bn ≥ n
→ n UNLESS Bn HAS TAG 01

A simple form of multiway switch is incorporated.

SKIP Bn INSTRUCTIONS

This causes control to be advanced so as to jump past the number of unconditional control transfers specified by Bn.

(v) Item directives.

The directive which precedes all hand coded routines and dictionaries, is

ITEM n

and additional entry points are denoted by

INDEX n = CC

Both directives cause the address of the following word to be entered in the index, and the former, in addition, limits the scope of labels to an item.

(vi) Subroutine calls.

In the hand coded language these take the form

CALL Rn and CALL SMALL Rn

The semantics of these instructions have been influenced by the Atlas extracode for subroutine entry,

1102, Ba, Bm, n

The specification of this extracode which is to be simulated on other machines, is

Ba = address of next instruction, →Bm

The B-register denoted by Bm contains the address of the DOWN sequence* which obtains the serial number of the routine to be entered from (Ba - 1).

The instruction signifying the dynamic end of a routine is

END

which transfers control to the END (or UP) sequence*.

(vii) Shifting operations.

These are intended only for transferring information digits to another part of a word via consecutive registers. They should not be used for circular shifting or be relied on for losing digits which overflow or underflow. The principal instructions in this class are

SHIFT Bn UP n , SHIFT Bn DOWN n

(viii) Peripheral Instructions.

The instruction for reading one character from the input stream is

Bn = NEXT CH /→n

---

* Ref. 4, p.41

It reads the character in Atlas Internal Code* to Bn, but at the end of a line or card, it transfers control to label n of the routine.

The output instructions enable characters (basic or composite), octal words and decimal numbers to be printed, and permit the layout to be influenced by spaces and new lines.

(ix) Constants.

For hand coded dictionaries and other miscellaneous purposes, integers and & words may be written into the program by means of the formats

n
-n
n + TAG 01

3.3

The Mercury program which translates the hand coded language into machine orders can now be considered in greater detail.

The syntax and semantics of the language are specified on a tape which is read by the primary phase of the program. The syntax of each instruction is specified by the instruction itself which is followed on a new line by its translation in machine code, finally terminated by the reserved symbol Ø, for example,

---

* Ref. 1, section 8.7.2

```
 Bn = (Bn + n)
0101, n1, n2, n3
Ø
(Bn) = Bn + n
0121, 99, n2, n3
0113, 99, n1, 0
Ø

    etc.
```

A subroutine of the program reads this material a section at a time, simultaneously performing an operation similar to that in the compiler compiler which converts characters with metasyntactical significance into their internal form. This consists mainly of changing the two-shift 5-bit paper tape characters into a 6-bit single shift code, and of ignoring erazes and spaces. Each format and its translation is written to a sector of the drum. The Manchester University Mercury computer with two drums, permits the use of up to 384 instructions, of which only about 180 are used.

In the translation phase of this program, instructions are read a line at a time by the subroutine mentioned above. This ensures that insignificant differences in punching between the instruction and its syntactical definition are removed in order to make recognition possible. The next stage is to scan the line, substituting the character n for each decimal integer, and to preserve these integers in a list for future reference. Since integers greater than $2^{22} - 1$ do not occur in the compiler compiler, it is possible to use un-rounded accumulator arithmetic (which is exact for integers less than 2^{29}) to evaluate and store these numbers.

If the instruction bears a label it is then removed from the line. The format and position of labels in the object program, and hence their treatment at this point, depend upon the nature of the target language.

The instruction is now in a form suitable for analysis. Sectors are read from the drum and compared to the instruction on hand until either complete recognition is made, or an empty sector is read (i.e. the instruction has not been recognised). In the former case, the translation is obtained directly from the words immediately following the recognised format and in the latter instance the unidentified instruction is printed out, together with the caption - UNKNOWN FORMAT, and a warning hoot.

Any of the 5-hole tape characters which is known not to appear in the target language may be reserved to modify the output. For example, the character Ø which terminates the semantics of an instruction is used to terminate the output of each translated format. If more types of modification are required than there are spare characters available, then one of the spare characters can be used to enter the sequence of instructions in the Mercury program at the label whose number follows that character. The symbol π is reserved for this purpose in the Atlas and IBM 7090 versions of the program for such purposes as optimizing the translation and simulating the extracodes which perform the general shifting operations on Atlas.

CHAPTER 4

A proposed scheme for using the compiler compiler on an IBM 7090 Data Processing System.

4.1

When the writing of the compiler compiler was almost completed some seven months before Atlas was sufficiently developed it was hoped to begin the experimental development of the system on an IBM 7090 computer with a main store of 32,768 words. Although the proposals outlined in this chapter did not materialise, they may serve as suggestions for subsequent adaptation to another suitable computer.

The IBM 7090 is a single address machine with standard word length of 36 binary digits, which is normally used for instructions, integers and floating-point numbers. In addition to the main store, there are three index registers (XR1, XR2, XR4) which are primarily used for address modification. In such cases the instruction is executed as if the address were the stated address minus the contents of the specified index register. The concept of address modification is extended for a large group of instructions by means of indirect addressing. When this is indicated in an instruction, the address is calculated in the normal manner, but instead of taking this word as an operand, the instruction interprets this word as the address of its operand. The operations of both fixed and floating point arithmetic

are performed with a single accumulator. The normal input is from 80 col x 12 row cards and output is either to a card punch or line printer. A full description of the IBM 7090 Data Processing System is to be found in ref. 5.

The principal input languages available for the IBM 7090 computer are FORTRAN, and FAP (FORTRAN Assembly Program), a mnemonic machine language. The 7090 FORTRAN Monitor System[*] makes it possible for a program to be written in both FORTRAN and FAP where it is necessary that the inner loop of a program should be as efficient as possible. This system has been used for the compiler compiler since FORTRAN simplifies the input/output subroutines and FAP is more suitable for the detailed coding of the major remaining part.

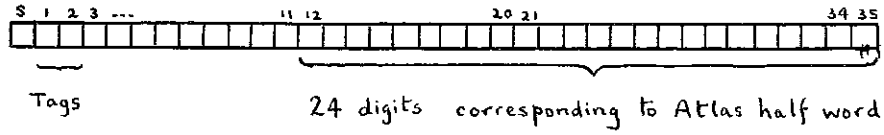
4.2

Since there are a large number of analysis records, dictionaries and lists in the compiler compiler, careful consideration must be given to the allocation of store to integers and addresses. In the Atlas version the instruction code makes it convenient to have two integers or addresses in a full 48-bit word and the possibility of doing something similar in the IBM 7090 version has to be considered. The

---

[*] Ref. 6, p.61

maximum length of the address of a word in the store is 15 binary digits; 2 digits are required for tags, making it possible to pack two such words into 36 digits. However, the 7090 instruction code contains no single instruction for accessing 18-bit words, making it necessary to use full 36-bit words for all integers and addresses. A 36-bit word has therefore been given the following structure.



Digit 's' is the sign digit, and digits 1 and 2 are used as the tags. When such a word is interpreted as an address, digits 1-20 are ignored.

A convenient method of loading a 36-bit word with integers which may or may not be tagged, is provided by the binary structure of certain instructions. The four instructions which correspond to the four different tags in digits 1 and 2 are

IOCD	000
TXI	001
TIX	010
TXH	011

The use of the core store is in principle, the same as for the compiler compiler on Atlas (ref.4, p.33), but since the one level store is of more limited size, the position of the object program has been moved and reasonable estimates of the size of record store and chain have been made (fig.3).

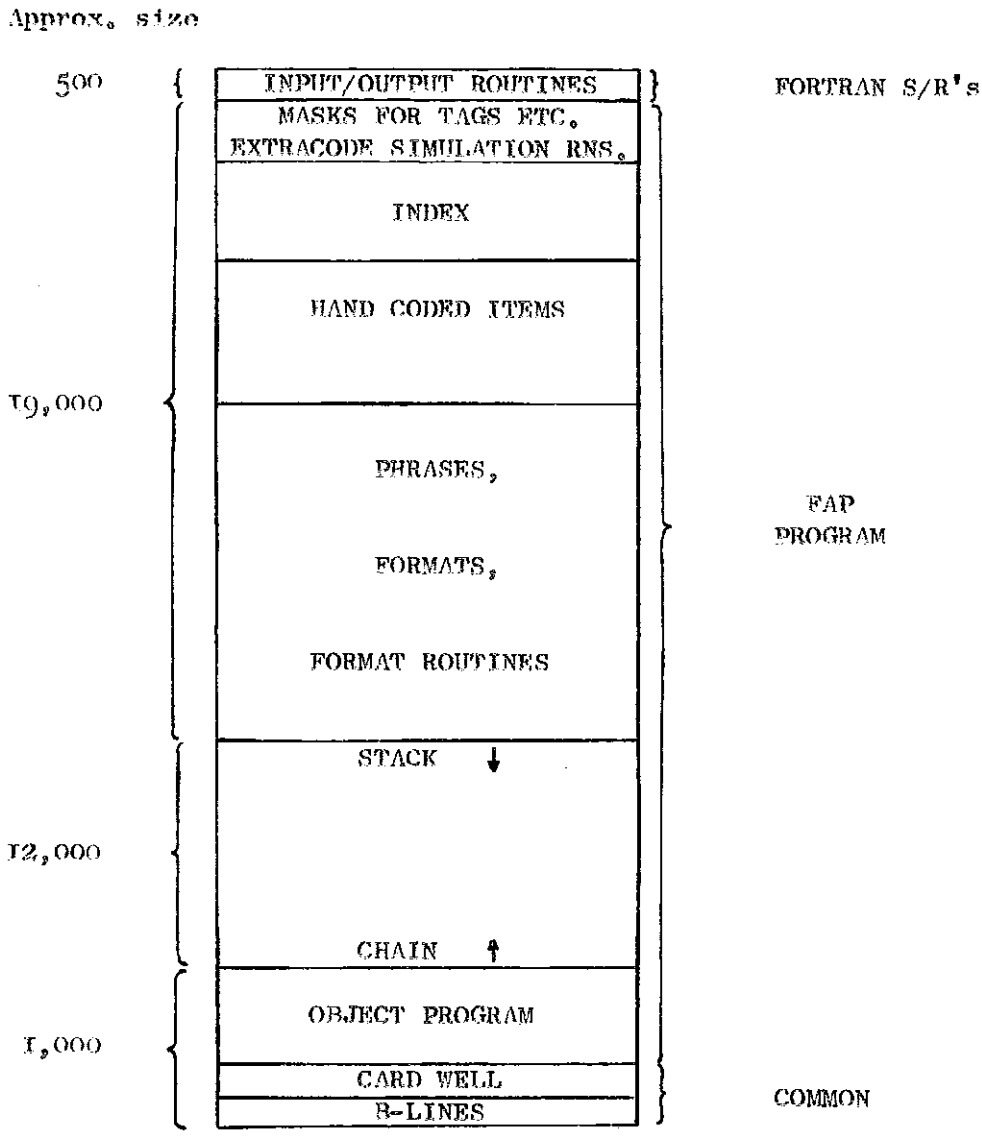


Fig 3. Storage allocation for IBM 7090 version of compiler compiler.

Whenever an extracode has been used in the Atlas version it calls a closed subroutine (generally in the fixed store,) to perform the desired operations. The extracodes which have been used extensively have been simulated in the IBM 7090 version by placing closed subroutines in a part of the store where they will not be moved by the deleting mechanism. The area which has been allocated to these and other routines which are more conveniently made 'store invariant', is at the front of the store, immediately before the index. First in this region are a number of FORTRAN routines for output and a subroutine of the line reconstruction routine which reads a card. Following these in this 'fixed' region is that part of the main FAP program which consists of the closed 'extracode' subroutines and a number of masks in fixed addresses which are used in the manipulation of tags.

Communication between FAP and FORTRAN routines within a program is generally made by means of COMMON storage. This is the region at the back of the store, beginning at 77461 (octal) and continuing forward through the store. In the compiler compiler, parameters are usually passed from one routine to another, by placing them in registers in the B store. By using the COMMON area for this purpose it is possible to use the B registers as parameters of the FORTRAN routines. The symbolic addresses of these registers are simply Bo, Bl, B2, ..., B127. The FORTRAN subroutine

which reads a card uses a further 72 registers in this region in order to pass on its output to the line reconstruction routine.

4.3

One feature of the compiler compiler which it is desirable to retain is the 'store invariance' of the hand coded items in the record store. This means that instructions causing control transfers within routines must depend on a relative distance rather than the absolute location of the instruction to which control is passed. This is achieved quite simply in the Atlas version, for the control is in B127, and the relative distance (which is computed at the time the instruction is compiled) is added to B127 to make the transfer. Since control transfers occur frequently in the compiler compiler it is important that the FAP coding of these instructions should be as efficient as possible. This has been achieved by calculating the address of the instruction to which control is to be passed, relative to the origin of the routine <u>at FAP assembly time</u>, and modifying this <u>at run time</u> by index register XR4 which is arranged always to contain the 2's complement of the address of the current routine. In this way, the single instruction

TRA 7,4

causes control to go to the instruction occupying the eighth

word of the routine.

An extra instruction, however, is required at every point where a routine may be entered, to set XR4, but since the number of subroutine calls is considerably less than the number of control transfer instructions, fewer orders are required for this method than for that mentioned in 1.3. There are in fact, only five instructions in the hand coded language, and one which the system compiles itself, which cause routines to be entered, namely, CALL Rn, CALL SMALL Rn, ENTER Bn, ENTER PR Bn, DOWN, and the entry to the TRANSPLANT sequence. The translation of each of these instructions, and the sequences which simulate the extracodes all ensure that XR4 is set to the 2's complement of the address of the appropriate routine.

The method of labelling the hand coded FAP items has been adopted to simplify the coding of the index. All the FAP material is treated as one FAP routine in order that symbolic labels may be used in the index to refer to an item in the record store. The first word of each item is effectively labelled by a right bracket followed by the item number (see fig.2) and the corresponding entry in the index consists of the instruction which corresponds to the appropriate tag, followed in the address part by the symbolic label. On translating the single FAP routine into machine language object program, the location of each item will automatically be entered into the address field of the

of the corresponding entry of the index. This system for labelling items is extended to provide a way of labelling instructions within items. Label m in routine n is written as m)n thus distinguishing it from label m in any other routine (see fig.2).

4.4

The Atlas extracode 1102, Ba, Bm, n is used in three different contexts in the compiler compiler, namely, the call for a large routine, the call for a small routine, and the entry to the TRANSPLANT sequence prior to calling the appropriate interpretive routine. It is better to have a closed subroutine for each case than a general one which would simulate the extracode more precisely because of the difficulty of specifying whether the DOWN or TRANSPLANT sequence is to be entered. The three cases are quite similar, and therefore the call for a large routine will serve as an example.

Translation of CALL Rn.

```
        TSX     BIGR, 4         jump to closed s/r. to
                                simulate extracode.  XR4
                                = 2's comp. of addr. of
                                next instn.

        IOCD    n1              parameter for DOWN seq.

        LAC     ORIGIN + n0, 4  XR4 = 2's comp. of addr. of
                                routine, for return from
                                called routine. (see Notes.)
```

Closed subroutine

```
BIGR    PXA     4               Acc = XR4

        ERA     =0777777777777  Acc = 1's comp. of XR4

        ADD     =2              Acc = link

        STO     B70             Set B70 = link

        LAC     ORIGIN + 239,4  XR4 set for DOWN seq.

        TRA     0,4             enter DOWN seq.
```

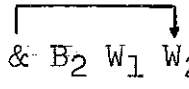
Notes:

(i) ORIGIN is symbolic address of the first location in the index of the compiler compiler.

(ii) n0 is replaced by the current routine's serial number by the Mercury translation program.

4.5

The planting of machine orders in the Atlas version of the compiler compiler is a fairly straightforward process since the function code resembles so closely the internal form of an instruction. The alphabetic function code of the IBM 7090 on the other hand, bears no more than a one-one correspondence with the pattern of digits representing the instruction. Also, the internal format of the instructions vary so that a type number has to be allocated to each instruction to enable it to be correctly assembled. A table, therefore, is provided which gives the function digits and type number corresponding to the mnemonic code of each instruction. The built-in phrase routine for the alphabetic part of a machine instruction then uses this table to construct an analysis record



in which W_1 is the word representing the function digits, and W_2 is the type number of the instruction.

The interpretive routines for planting and obeying FAP machine instructions use a common sub-routine which assembles the instruction in a 36-bit word. This subroutine examines W_2 to determine the type of instruction on hand, and plants the address, tags, flags, counts and decrements in the correct digits.

CHAPTER 5

Development techniques

5.1

It was intended as part of the Manchester University Atlas project, that the Mercury Autocode Computing Service should be transferred to Atlas at the earliest possible date. The general implication of this was that the Mercury Autocode compiler, and hence the compiler compiler, would have to be tested and developed on Atlas during its various stages of commissioning. As different parts of the computer came into use it became necessary to make changes in the machine coding of the program, which confirmed the desirability of the machine-independent language of the hand coding and its flexible translator. In this chapter, the general approach to the testing and development of the compiler compiler is discussed, and an account is given of the manner in which the temporary deficiency of hardware on Atlas at different stages was overcome.

5.2

In section 2.4 the Atlas version of the compiler compiler was divided into three phases, associated with the three forms of input used. The first phase, written in Atlas Octal Input, consisted of an input program which enabled the second phase to be input in a more sophisticated language, which in turn allowed the remaining part to be written and input in the language of the compiler compiler.

Errors occurring in the last of these phases present no problem, for the incorrect statements can simply be corrected and recompiled as they stand, but those occurring in the octal section are generally less conveniently rectified. If the correction can be made in its own space then the relevant changes can easily be worked out in octal, but when extra orders are required, control transfer instructions throughout the routine may be affected, and the location of each of the following items will be altered accordingly.

One of the advantages of having a simple input routine embedded in the compiler compiler is that together with the routine for deleting an item, it simplifies the correction of all kinds of errors in the program which it reads. In order to amend an item in the record store the faulty version is deleted (i.e. the store which it occupied is recovered) and the corrected version is read in the normal manner. This procedure is quite satisfactory so long as the faulty item is not one of those which enables the Item routine or the deleting routine to function. It is conceivable that one of these items is found to contain an error which does not affect the simple path used by the Item routine and its replacement is required. In order to replace these items (and others too,) an item replacement routine (IRR) which uses the Item routine and the deleting routine as subroutines has been included in the system. This routine reads the

new version of the routine to the record store and allocates it a specially reserved serial number to prevent any attempt by the rest of the program to use it before it has been assembled. Once it is complete the serial numbers of the old and the new versions are interchanged, and the old version (with the reserved serial number) can safely be deleted. Certain precautions must be taken when replacing an item.

(i) It must be in the record store. No facility has been provided for replacing chain dictionary.
(ii) The IRR itself and the top-level 'master routine' may be replaced, but in general the program will not be able to continue in these cases. The IRR, therefore, in these instances ends with a loop stop, and has to be restarted manually.

The IRR is entered, in the same manner as the Item routine, upon recognition of the master phrase REPLACE ITEM.

5.3

A simple method of correcting mistakes in the program is valuable in the development of a large system such as the compiler compiler, but of equal importance is an efficient means for diagnosing errors, or rather for printing those parts of the store which enable the error to be located. Two diagnostic routines which are part of the system itself

have been extended to meet the special needs of developing various parts of the program. In many of the routines in the compiler compiler, checks are made to ensure that the material being processed is legal; a fault is identified by the fault number and the serial number of the routine in which it occurs. The faults which are detected reveal logical errors in the user's compiler and illegal statements in the source language program. Whenever a fault is discovered, one of the diagnostic routines is entered, depending on the nature of the fault. If the error is such that compiling (of compiler or source program) may continue, the fault is monitored and the program proceeds. The output at this point gives the fault number, routine number and the line number of the line of input which caused the fault, in the following format.

COMPILER COMPILER FAULT n R n L n

If the fault is catastrophic and further compiling is impossible, then the monitoring above is printed out together with the values of all the non-zero B registers.

Although the information above is useful in the diagnosis of faults it is often insufficient and further store lines are required. The catastrophic fault routine is therefore extended to print any specified region of the store in octal, with the address of every eighth word. A number of different versions of this routine have been made, which print the main stack and various parts of the record store and chain.

It is helpful sometimes to know not only in which routine a fault occurred, but also which routines the program had been in before the fault was discovered. When this is required, modified versions of the DOWN and END sequences can be inserted by the IRR, which print out the serial numbers of routines as they are entered.

Two routines of the system allocate serial numbers to newly occurring phrases and formats. In order to diagnose faults concerned with phrases, formats and format routines which have been read by the system it is advantageous to know the appropriate serial number. For this purpose, whenever a new serial number is allocated, the phrase or format is printed out together with its serial number.

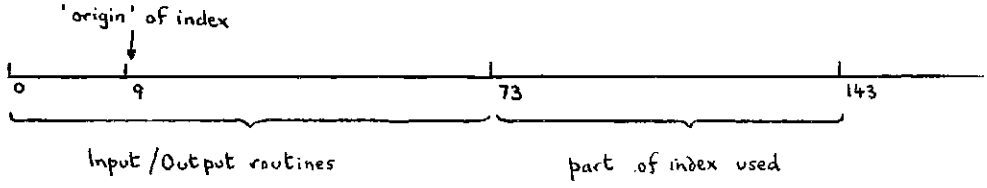
5.4

In March 1962 the subsidiary store from the prototype Atlas (1024 words) was temporarily connected to the Manchester University Atlas to enable preliminary testing of the central computer to begin, and some of the system programs to be developed. Since the first 8,192 words of main core store were not to be working until October 1962, it was useful to commence testing, if at all possible, on the 1024 available words of store using interrupt control. It was concluded therefore, that that part of the compiler which would enable the Item routine to function should be tested, and having done this, to test certain other important routines, whose subroutines and working space would fit into the small store.

It was possible to make fairly thorough tests of the Item routine, the ERR and the two dictionary routines[*] which gave little or no trouble when they were later used in the main store.

For these tests extracodes were not available and it was necessary to write two routines to read one character from 5-hole tape, converting it to Atlas Internal Code, and to punch on 7-hole tape a 24-bit half-word in octal followed either by a space or a new line. These two routines use the same B-lines as the extracodes (B91-B99) and have been written so as to occupy only 73 instructions. These sequences are complicated by the hardware which is provided for the time-sharing of peripherals and such things as stopping and starting the reader have to be programmed.

The serial numbers of the routines used in this phase all lie in the range 130-266 which means that the index need occupy only 69 48-bit words. However the program assumes that the origin of the index (i.e. index 0) lies in the store, and so the first 65 unused full words of the index are overwritten with the last 65 words of the input/output routines.



[*] Ref. 4, p.34 f.

In order to reduce the number of instructions still further, some routines are reduced to the few orders which will actually be obeyed in the simple tests and two other routines which are called in, but have no effect on the program, are removed and replaced by a dummy routine which simply returns control to the calling routine via the END sequence. Incorporating all these modifications sufficient store remains in which to assemble an item of about 12 instructions.

Apart from the difference between the size of the subsidiary store and the main store, there is a difference in the way the central computer treats instructions from the two stores. In order to prevent programmers from making inadvertent references to a store other than the one in which the instruction is located, by means of B-modification, the presumptive address (i.e. the address part of the instruction) and the effective address of an instruction must, in general, refer to the same store. In the compiler compiler, whenever a store reference is made, it is generally by means of an address which is in a B-line. For example, if B78 is to be planted in the next available half-word of record store (whose address is in B88) then the instruction

0113, 78, 88, 0

is the natural one to use. However, the presumptive address refers to the main store, and the effective address refers to the subsidiary store. The origin of the

subsidiary store is in octal location 7000 0000, and therefore this number should be subtracted from B88 before the order is obeyed and the order rewritten

0113, 78, 88, 7000 0000 (octal).

All store addresses then, are initially made relative to the origin of the subsidiary store and instructions referring to the store have 7000 0000 added into the address part. The latter modification is conveniently made during the translation of the symbolic machine language into Atlas Octal Input.

5.5

As soon as the first 8,192 words of main core store became operational, it was essential to transfer the development of the compiler compiler to this store even though much of the hardware and fixed store programs was not yet working. Interrupt control was still the only practical control to use, so extracodes could not be used. The only extracode for which substitution cannot easily be made, is that which is planted in front of an analysis record in a format routine. It was therefore possible to develop the entire primary phase of the compiler compiler whilst on interrupt control and to postpone the change to main control until operation on this control was more certain. Later, when main control became available, it was possible to use some extracodes (including the one planted in format routines) but full operation under the Atlas Supervisor had

to be left until a later date. Thus, the system was developed on the main store in three phases.

(i) Interrupt control; no extracodes.

(ii) Main control; certain extracodes.

(iii) Main control under supervisor; all extracodes.

Throughout the first two phases, the subsidiary store was employed for all the extracode simulation routines and for certain other operations which the supervisor would perform later. The following routines were written.

(i) Read one character in Internal Code from 5 or 7-hole tape.

(ii) Punch one character in Internal Code on 7-hole tape.

(iii) Disengage reader and disengage punch.

(iv) Set up page address registers.

(v) Change from interrupt control to main control.

(vi) Entry sequence to catastrophic fault routine.

Notes:

a) The input/output routines were made compatible with the extracodes which would later replace them.

b) The sequences for disengaging reader and punch are used as a means of controlling input and output. The master phrase END OF MESSAGE appearing at the end of all tapes read by the system, causes the reader to disengage, and the tape to stop. Subsequent tapes are entered by re-engaging.

c) Due to the 'store-invariance' of all routines, the location of the fault routine is uncertain. The sequence (vi) above enables it to be entered manually should the need arise.

One of the disadvantages of the bootstrap procedure is that much of the material is in the symbolic machine language which is not a very compact form of input. As it is essential to keep input to a minimum, a more compact form of input is achieved by means of a binary loader and binary output routines located in the subsidiary store. This makes full use of 7-hole tape, reducing to about 25% the time required to input a given section of the program. The representation chosen used eight tape characters to represent a 48-bit Atlas word. The most significant tape digit is chosen as a parity digit and makes each character of odd parity (allowing a binary tape to be reperforated on the Flexowriters in the laboratory). In addition to the parity check, a 4-character check-sum* is punched at the end of each binary block. Blocks can be of any length, and may be written to any part of the main or subsidiary stores. At the beginning of each block there is a block marker, a 4-character address specifying the location of the first word of the block, and a 4-character number denoting the length of the block. About six inches of Flexowriter upper case characters separate the blocks.

---

* The check-sum is calculated by simple addition. The technique of using 'end-around-carry' is unnecessary since the most significant digits are parity-checked.

Binary output normally commences at the origin of the index and continues in blocks of sixty-four 48-bit words until the block containing the address in B88 (the head of the record store) has been punched. At this point an 'end-of-tape' marker is punched, the reader and punch are disengaged, and control is passed to a routine which checks the tape against the store. Should the tape fail to correspond to the store, the reader is disengaged and the block is punched from store again. On re-engaging the reader, checking is resumed until the 'end-of-tape' marker is reached and any blocks which have been punched for a second time can be checked again. In this way a convergent process is established in which the number of incorrectly punched blocks becomes zero.

The most serious disadvantage of the method of development which has been described, is in the length of the binary tapes which have to be input before each test can be run. If a test ends in an unexpected manner, there is no guarantee that the program has not been overwritten at some stage, and therefore the whole program should be read again before the test may be repeated or further tests run. Even with the binary loader, paper tape input is prohibitively slow. When magnetic tape became available on Atlas it was possible to use this for the same purpose as the binary tapes had been used for, and the time taken to write the program

to the main store was reduced from minutes to a matter of seconds thus increasing the amount of testing which could be done in a given time.

CHAPTER 6

Primary Assembly Routines.

6.1

In section 2.3 reference was made to routines which compile a direct translation of certain instructions occurring in format routines, instead of planting analysis records which are later to be interpreted whenever the format routine is obeyed. In the literature on the compiler compiler, these routines have been referred to as 'primary assembly routines' and 'primary compiling routines', but since each one is generally associated with an interpretive routine for the same instruction, when referred to in this context, they are sometimes called 'compiling versions'.

The compiling versions are not fundamental to the compiler compiler inasmuch as a compiler could be generated without them, but their chief purpose is to produce a more efficient compiler. To illustrate this, consider the sequence of Atlas machine orders to be obeyed when the analysis record for the built-in instruction

$$\beta_{88} = \beta_{88} + 1$$

is 'obeyed'. The analysis record of this instruction is 17 half-words long and the TRANSPLANT sequence which copies it to the main stack, obeys about six machine orders for each word of the analysis record. A further 25 orders are obeyed in going down to the interpretive routine for [AB]= [WORD]

which itself takes about 80 orders to interpret the record, making a total of the order of 200 machine orders whereas a single machine instruction planted by a compiling routine would have the same effect. From this it can also be seen that a considerable amount of space can be saved in some circumstances. The analysis record above is preceded by an extracode and is finally rounded up to make it occupy an even number of half-words. In this example, nine 48-bit words are saved by the compiled form of the instruction. The space which the primary assembly routines themselves occupy is only of secondary importance, for they can be deleted if desired, once the compiler has been assembled, vacating the store which they occupied for the use of object programs.

In one sense, the compiling versions belong more to the compiler which is generated than to the compiler compiler itself, for it is in the format routines of the former that the greatest efficiency is required. The choice of compiling versions, and even the coding of them, may depend to some extent upon the type of instructions which occurs with greatest frequency in the format routines of the compiler. However, compiling versions of most of the built-in instructions have been provided since it is expected that the compiler writer will use these fairly frequently in his format routines. If he repeatedly uses some auxiliary statements for which compiling versions would increase efficiency appreciably, then he is recommended to write his

own on the lines of those which have been included in the compiler compiler. (See Appendix 1, p.38f. and Appendix 3.)

Although the length of the compiling versions themselves is not of primary importance, it is nevertheless undesirable that they should be any longer than necessary, especially as there is a simple way of shortening them. By reading them in a second time, and deleting the first copies, they compile themselves, making them both shorter and more efficient.

Not all the built-in instructions result in compiled sequences which are shorter than the corresponding analysis record; for instance the instruction

[AB] = NUMBER OF [PI]

requires 10 compiled instructions whereas its analysis record occupies only 7 words. Some compromise therefore, has to be made in deciding which instructions are worth compiling for sake of efficiency, and which should be interpreted in order to conserve space. This is considered in some detail for each type of instruction in the following section.

6.2

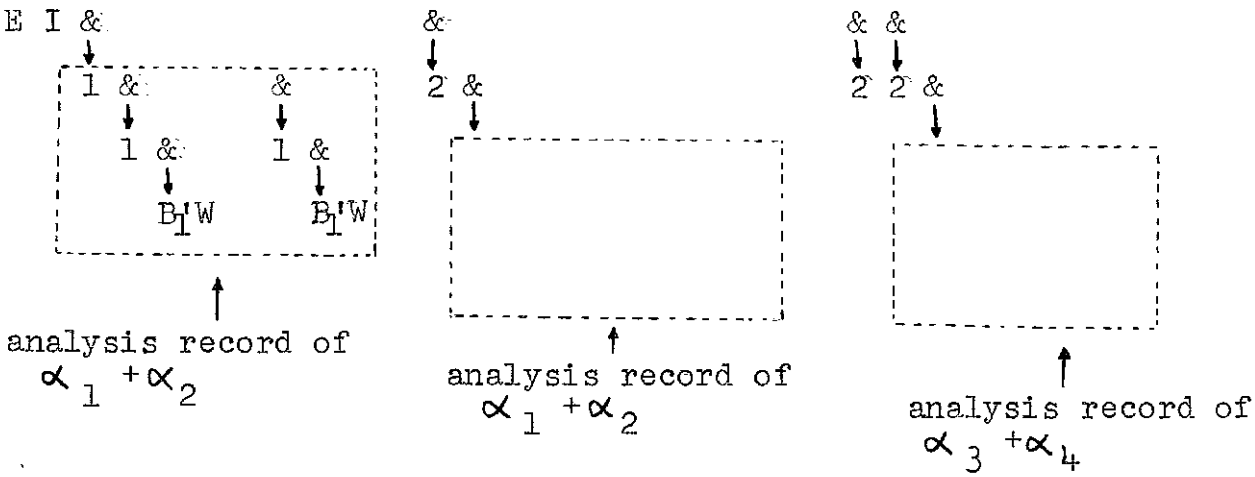
The built-in instructions conveniently fall into six groups within which the usefulness of compiling versions for each type of instruction is discussed. The effect of most of these instructions is described in appendix 1, p.10 ff.

(i)
[AB] = [WORD]
[AB] = [WORD] [OPERATOR] [WORD]
([ADDR]) = [WORD]
([ADDR]) = [WORD] [OPERATOR] [WORD]

An illustration has been given above of the advantage in compiling a simple instruction in this group; an example of a more complex instruction is now given in detail. Consider

$$(\alpha_1 + \alpha_2) = (\alpha_1 + \alpha_2) - (\alpha_3 + \alpha_4) \quad .$$

The representation of this instruction as a tree structure is,



which occupies the space of 23 full words.

The compiled instructions for the same are

```
0101, 83, 72, 1 + m
0104, 83, 72, 2 + m
0101, 82, 83, 0
0101, 83, 72, 3 + m
0104, 83, 72, 4 + m
0101, 84, 83, 0
0120, 84, 82, 0
0121, 82, 84, 0
0101, 83, 72, 1 + m
0104, 83, 72, 2 + m
0101, 83, 83, 0
0113, 82, 83, 0
```

(where m is the number of LSE entries)

which are 12 words shorter than the analysis record.

Not all instructions of this class are compiled. Those involving the third category of [ADDR] , namely [AB] (+) [ABN] are interpreted since they do not occur very often,

and the planting of a loop of instructions to pass along a chain list is not considered worthwhile.

A few frequently occurring instructions such as

$$[B] = [B] \pm [N]$$

$$[A] = ([A] + [N])$$

which would have inefficient translations if treated generally, are singled out at the start of the routine for [AB] = [WORD] and more efficient translations are specially compiled. (See appendix 3.)

(ii) [FD][COMMA][WORD][COMMA][WORD][COMMA][WORD]
PLANT [FD] [COMMA] [ABN] [COMMA] [ABN] [COMMA] [WORD] IN [B]

The compiling version for the first of these instructions simply plants an instruction which is obeyed when the format routine being assembled is used; the second plants instructions for compiling Atlas machine orders. Often when these instructions are used, the B-registers and the address parts are explicitly stated, and it is a simple matter to compile them. When an α or a β appear, then instructions would have to be planted to evaluate the value of the variables, assemble the instruction and plant it. For this reason only those instructions where the B-registers are explicitly stated are compiled. The advantages of compiling in these cases is again shown by comparing the tree with the compiled orders for the instruction

PLANT 0121, 127, 83, 2 IN B_{88}

The tree structure is

```
E   I    &     &     &     &     &
         ↓     ↓     ↓     ↓     ↓
     0504 0000 3 &   3 &   5 & & 88
      (octal)   ↓     ↓     ↓ ↓
               127   83    2 2
```

whereas the compiled orders are

```
0121, 82, 0, 0507 7723 (octal)
0113, 82, 88, 0
0121, 82, 0,  2
0113, 82, 88, 1
0124, 88, 0,  2
```

which are 6 words shorter.

(iii) [JUMP] [LABEL]
[JUMP] [LABEL] [IU] [WORD] [COMPARATOR] [WORD]
[FD], [WORD], 0, L [LABEL]

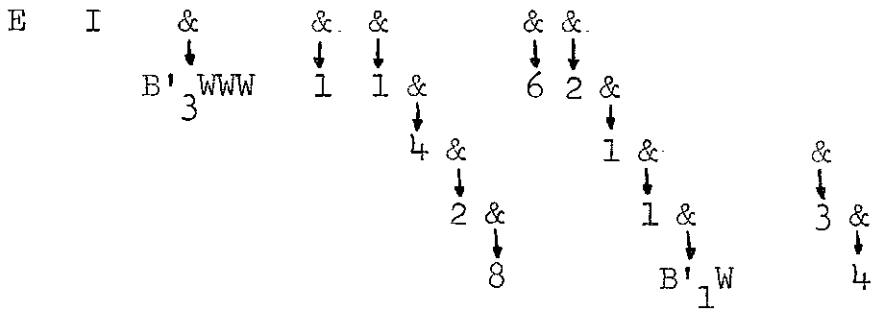
Most cases of →[LABEL] take the form →[N] and these can be compiled into a single instruction. The label reference is added to the list of label references which the routine assembly routine uses, and the value of the label is filled in at the end of the format routine being compiled, by the routine assembly routine. Where the label is an α or a β, instructions are planted which call in a small routine of the system which consults the label directory at the end of the routine in order to make the control transfer.

The conditional control transfer, and the Atlas machine instruction are compiled only if the label is an [N] and in the latter case [WORD] is again restricted to [N]; otherwise the instructions are interpreted.

Consideration of the conditional jump instruction

$$\rightarrow 6 \quad \text{IF} \quad \beta_8 \geq (\alpha_1 + 4)$$

shows the efficiency of the compiled instructions.




```
0101, 83, 72, m + 1
0124, 83,  0, 4
0101, 83, 83, 0
0170, 83,  8, 0
0226,127,127, 6
```

(where m is the number of LSE entries. The address part of the last instruction is filled in by the routine assembly routine.) 12 words are saved each time such an instruction is compiled.

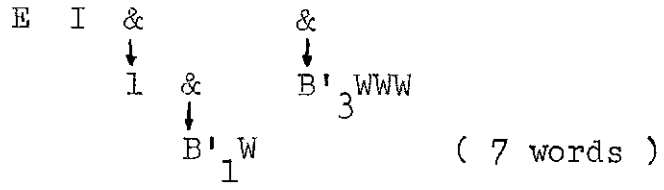
(iv)
```
[AB] = NUMBER OF [PI]
[AB] = CATEGORY OF[PI]
[AB] = ADDRESS OF [PI]
[AB] = CLASS OF [PI]
```

The compiling version for the first instruction of this group plants a loop of orders which counts the number of recursions in the structure of [PI]. In the next two, faults are indicated if the identifier [PI] is illegal, control being transferred to the fixed location (octal) 2001 3760 from which the fault routine is entered.

The two forms of a typical instruction in this group, namely

$$\alpha_1 = \text{ADDRESS OF } [A]$$

are analysed as before.




```
0101, 82, 72, n
0172, 82,  0, 0
0224,127,  0, 2001 3760
0121, 83, 82, 0
0113, 83, 72, m + 1
```

(where n is the LSE number of [A])

```
(v)    LET [PI] [EQV] [RESOLVED-P]
       [JUMP] [LABEL] [IU] [PI] [EQV] [RESOLVED-P]
        LET [PI] = [GENERATED-P]
       [JUMP] [LABEL] [IU] [PI] = [GENERATED-P]
       [PI] = [AB]
```

It is generally too complicated for it to be worthwhile compiling machine orders to perform parameter testing or resolving instructions. However, a fairly common type of instruction in this group can be compiled without too much difficulty. It is in those cases of the first two instructions of this group where the analysis record of [RESOLVED-P] with respect to [PI] is only one level deep. The analysis record for [RESOLVED-P] in this case consists entirely of P- words.[*] The compiled form of these instructions is

---

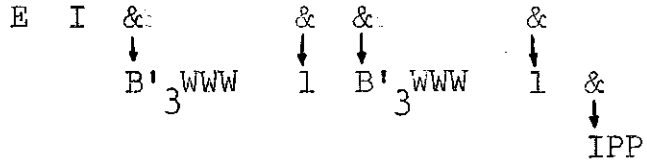
[*] Ref. 4, p.37

generally longer than the corresponding tree structure but the efficiency of the former makes them worth including.

The instruction

→ 3 IF [ADDR] ≡ [AB] + [ABN]

has an analysis record



which occupies 11 words; 12 compiled orders are required (- for details, see the program in appendix 3).

(vi) The remaining built-in instructions are of less interest, and the details of the compiling versions are obvious from the program in the appendix.

CONCLUSION

The main purpose of the compiler compiler, as stated in the Introduction, is to simplify and shorten the implementation of compilers on automatic digital computers. At the time of writing, a compiler for Mercury Autocode is in the course of being developed on Atlas.

Initial testing of the compiler compiler began in March 1962 and continued for about a month while only the subsidiary store was available for programs. It was not until October 1962 that the first 8,192 words of main store were working. During the following 3 months, the program was developed in the three phases described in chapter 5 in order to make maximum use of the hardware as it was commissioned. However, the machine was not available for the whole of that time since other sections were still being commissioned. On 31st December 1962 when only 16,384 words of store could be used, the first simple Mercury Autocode program was compiled and obeyed. Less than a week later, most of Mercury Autocode with the exception of matrix, double length and complex facilities were tested and working, having used less than 20 hours useful machine time.

It is hoped that Atlas Autocode, Extended Mercury Autocode, ALGOL and other compilers to be implemented on

Atlas in the future, will require even less machine time for development using the Atlas Supervisor and the 'one-level store'.

REFERENCES

1. Ferranti Ltd.

Atlas Reference Manual.

(This document has not been published generally. It constitutes a description of the system from the view point of a knowledgeable programmer and is primarily intended for the use of persons concerned with the design of the system.)

2. Kilburn, T., Payne, R.B., Howarth, D.J.

The Atlas Supervisor.

Paper presented at the E.J.C.C.,

December 1961.

3. Brooker, R.A., Morris, D.

Some Proposals for the Realization of a Certain Assembly Program.

The Computer Journal, Vol.3, p.220.

4. Brooker, R.A., Morris, D., Rohl, J.S.

Trees and Routines

The Computer Journal, Vol.5, p.33

5. IBM Reference Manual

7090 Data Processing System (A22-6528-1)

6. IBM Reference Manual

709/7090 FORTRAN Programming System (C28-6054-2)