THE IMPLEMENTATION OF

ALGOL W

ON A

BURROUGHS B6700 COMPUTER

by

Henry D. Meekin

A thesis presented in partial fulfilment

of the requirement for the degree of

Master of Science in Computer Science

at

Massey University

May 1976

ABSTRACT

This thesis describes an implementation of a revised version of ALGOL W on a Burroughs B6700 computer, and was written so that excerpts can be made to produce a user manual and a system documentation manual. The first part is a brief discussion of the language as implemented and discusses the main features of the language and the differences from ALGOL 60. The remainder of the thesis gives a detailed description of the compiler.

ACKNOWLEDGEMENTS

In presenting this thesis I would like to take this opportunity to express my thanks to the following people:

To my supervisor, Lloyd Thomas, whose guidance and encouragement helped at all times.

To Neil James for helping with those niggling program errors.

To the Computer Unit operators, especially Colin Read, for performing their job so well in helping me run my program.

Finally to my family and the group, for their persistence in prodding me to finish.

Massey University Harry Meekin

May 1976

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 0

INTRODUCTION

The language ALGOL W was first described in a report drafted by Niklaus Wirth (hence the 'W'), asked for by IFIP Working Group 2.1 at its May meeting at Princeton, 1965. The report was distributed to members of the group as a "Proposal for a Report on a Successor of ALGOL 60" [1]. However, at its October meeting, 1965, at Grenoble, the group decided the report did not represent a significant advance on ALGOL 60 so it was dropped as an official Working Document of the Group. Wirth then collaborated with C.A.R. Hoare and revised and supplemented the draft. This revised report can be found in Wirth and Hoare [2].

Why then was it felt to be of sufficient interest to implement ALGOL W? ALGOL W is similar in many aspects to ALGOL 60 but some concepts have been simplified and some extensions have been introduced. The most important extension is the introduction in the language of the concept of generalised data structures. To supplement the array concept, which is virtually unchanged from ALGOL 60, there is a new data structure, the record ([3] and [4]). This makes ALGOL W a more powerful language than ALGOL 60 in its ability to handle data structures and therefore a more suitable language for use in the commercial field.

The language described by Wirth and Hoare, with a few changes [5], was first implemented on an IBM 360 at Stanford University [6] in 1968.

In 1971 further revisions were made to the language at Stanford [7] and an improved implementation was developed. ALGOL W has since gained only slowly in popularity being used mainly at University sites but also at a few commercial installations.

This thesis describes an implementation of ALGOL W as revised in 1971, with some further modifications to be described in subsequent chapters, on a Burroughs B6700 computer. Chapter 1 gives a brief description of the language as implemented. Subsequent chapters describe the organization in detail of the compiler.

This thesis has been written so that excerpts may be made to produce both a user manual and a system documentation manual.

Chapter 1

BRIEF DESCRIPTION OF THE LANGUAGE

As the name suggests, a large part of the language is taken directly from ALGOL 60 [8]. As ALGOL 60 has been in extensive use for some time and is therefore fairly well known, this chapter will only discuss the major changes to ALGOL 60. A full description of the Extended ALGOL W implemented by the author may be found in Appendix B.

1.1 Data Types

As is the case in most modern languages there has been an increase in the number of primitive data types from the three in ALGOL 60. The types included in the language are integer, real, long real (double precision real), complex, long complex (double precision complex), logical (equivalent to ALGOL 60's boolean), bits, string, and reference. The ALGOL 60 concept of own variables has been dropped as it doesn't add any power to the language and leads to semantic ambiguities in many cases.

The type complex is internally represented as two real words, one for the real part of the complex value, and one for the imaginary part. The type long complex is internally represented as two long real words with the same meanings as in the case of complex.

The type bits is one word containing a sequence (i.e. an ordered number of elements) of binary digits. Operations defined for bit sequences include the logical operations $\neg$, $\wedge$ (and), and $\vee$ (or), and those of shifting left (shl) and shifting right (shr).

To represent an ordered sequence of characters the type string has been included. When this type is declared, the maximum number of characters able to be held in the sequence must be

explicitly stated, e.g.

string (10) A

declares a variable A which represents a character sequence of up to 10 characters. The string type is internally represented in an analogous way to the Burroughs Extended ALGOL EBCDIC arrays. Originally the operations for string sequences included the catenation operator cat, but with the addition of the concept of substrings the cat operation was found to be redundant and so was abandoned.

The type reference will be discussed in section 1.4.

An interesting and very useful aspect of the design of the language is that the type and length of the result of evaluating every expression or subexpression can be uniquely determined at compile-time, so no type testing, except possibly on procedure entry, is needed at run-time, thereby not wasting execution process time for non-compatible type testing.

With the increase in the number of data types there is a greater number of possible type conversions. Automatic type conversion (i.e. conversion performed by the compiler) is confined only to cases where no confusion about the required conversion is possible, i.e. from integer to long real, and real to complex, and from shorter to longer variants of the types. All other conversions must be programmed explicitly by the programmer with the use of standard functions. This is so the programmer knows exactly what type of result he is getting rather than relying on a default conversion which he may only have vague or even mistaken ideas about.

1.2 Statement Sequencing

The concept for control of statement sequencing has been simplified. The switch declaration, switch designator, designational

expressions, integer labels, and label parameters have all been abolished.

The switch declaration and switch designator have been replaced by the case statement and case expression. These allow the selection and execution (or evaluation) of one of a list of statements (or expressions) due to the value of an integer expression. As the case construction is in use in a number of modern languages and is fairly well known, the reader is referred to Appendix B for more detail if required.

A goto statement can not lead from outside into any if statement, case statement, or iterative statement.

There are three types of iterative statements:

(i) for<id>:=<for list>do<statement>,

(ii) for<id>:=<int.exp.>step<int.exp.>until<int.exp.>do<statement>,

(iii) while<log.exp.>do<statement>.

These are the simple and most common cases of iterative statements and more complex cases can be easily dealt with by explicit program instructions using labels. There are a few points to notice. The <for list> and step-until parts can no longer be mixed. The "step <int.exp.>" is optional and if missing a default step of 1 is used. The <id>, called the control identifier, is implicitly declared at the start of the for statement and is undefined outside the scope of the for statement. No explicit assignments are allowed to be made to the control identifier.

1.3 Procedures and Parameters

There are a few changes towards clarification and efficiency of implementation, to the ALGOL 60 concept of procedures.

The meaning of parameters is unchanged, i.e. they can be

explained in terms of the "copy rule". In addition to the "name parameter" and "value parameter" there has been added the concept of a "result parameter". This formal parameter, like the value parameter, can be thought of as a local variable. Upon the termination of the procedure the actual parameter, which must always be a variable, is assigned the value of the pseudo formal parameter.

Array parameters can only be called by name.

An actual parameter may be a statement (or expression) providing the corresponding formal is procedure (or <simple type> procedure). This statement (expression) is considered as a proper procedure body (function procedure body) without parameters. This enables a procedure (function) to be specified in the actual place it is to be used rather than in the head of an embracing block.

As mentioned in 1.2, the label parameter has been abolished. This results in no loss of power because the result of the old label parameter can be achieved by writing a goto statement in an actual parameter position as outlined in the preceding paragraph.

In this implementation the concept of virtual parameters has been included. A programmer may optionally specify parameters of formal procedures and thus enable compile-time formal procedure parameter checking. In the cases that this facility is used, no run-time parameter checking is needed on procedure entry. This concept is used extensively in ALGOL 68.

The specification of all formal parameters (except parameters of formal procedures) and the correct matching of actuals to formals has been made compulsory. The number of dimensions of an array parameter must also be shown. The specifications are included in the formal parameter list rather than in a separate specification part as in ALGOL 60. This is a much tidier form of specification as it has the attributes of the parameters in their actual position.

1.4 Data Structures

The only changes to the ALGOL 60 array concept are notational. The type of the array must always be specified and only arrays of the same type and dimension may be contained in the same array declaration.

There has been a major addition of another type of data structure, the record [4]. Like the array, records consist of one or more elements (called fields), but unlike the array the fields do not have to be of the same simple type, so that each field may occupy a different amount of storage. Because of this to select a particular field a computed ordinal number cannot be used. Each particular field is given a name (identifier) which is used in the program whenever that field is referred to. Also, unlike arrays, records are created dynamically by statements in the program rather than by declarations (see Appendix B, section 6.6).

The normal data types (see 1.1) are sufficient to represent the properties of the fields of records, but a new type, reference, is required to represent relationships between the records. For example, if a record which represents a person has a field named "father", then this is likely to be used to contain a reference to the record which represents that person's father.

References are also used to provide programmers access to records. For this purpose, variables of type reference should be declared in the head of an embracing block, e.g.

reference(<record class list>)<id list>.

The <record class list> is a list of the record classes to which the identifiers in the <id list> may refer. Thus reference variables are somewhat analogous to a restricted form of pointer variables.

A record class is a group of records which are similar, i.e. records

which have the same number of fields and corresponding fields have the same names and types. Each record class is introduced in a program by a record class declaration which associates a name with the class and specifies the names and types of the fields of that class.

So that any particular field of a particular record of a record class can be referred to, a field designator must have associated with it a reference expression which is a reference to the required record (see Appendix B, section 6.8). This is checked for compatibility at compile-time.

Because a reference variable may refer to more than one record class, it is sometimes necessary to know at a particular part of a program to which class the reference is then referring. To achieve this knowledge there is a logical expression,

<reference primary> is <record class identifier>,

which is true if the reference primary is referring to a record of that record class and false otherwise.

There is also a null reference, null, which points to no record, i.e. if a reference has the value null it is undefined.

1.5 Block Expressions

There has been the introduction of a typed block which is a block that has a value (see Appendix B, section 6). The block acts like a function procedure body with no parameters and is a useful notational convenience because like the statement parameter, it allows the function to be specified actually in the place where used, rather than disjointly in the head of an embracing block.

1.6 Assert Statement

During the running of many programs it is useful to terminate any further execution of the program if some condition arises.

This is achieved by the use of the assert statement,

assert <logical expression>,

(see Appendix B, section 5.1) which will terminate the program if the <logical expression> is not true.

1.7 Input/Output

In the original implementation of the language input/output was achieved by the use of primitive standard procedures READ, READON, READCARD, WRITE, WRITEON, and IOCONTROL [7]. These procedures did not allow for programmer formatting. In 1971, the University of Manitoba developed format-directed input/output facilities for ALGOL W [9]. Upon study of these facilities it was found that they were not as versatile as those employed by Burroughs Extended ALGOL [10 and 11]. Because of this and because of the desire to make the implementation compatible with the existing system on the Burroughs B6700, a slightly simplified version of the input/output facilities used by Burroughs Extended ALGOL [12] was adopted. These facilities include the file statement, read statement, write statement, space statement, rewind statement, seek statement, close statement, and lock statement. It is planned to add the standard ALGOL W input/output procedures so that this implementation is compatible with other ALGOL W implementations.