Context sensitive editing as an approach to incremental compilation

L. V. Atkinson, J. J. McGregor and S. D. North

Department of Computer Science, University of Sheffield, Sheffield, S10 2TN, UK

An incremental compiler for a block-structured language has been implemented. The system performs minimal recompilation in response to changes made to a program. To achieve this, a context sensitive editor was designed. This paper describes the internal representation of the program and gives a detailed account of the editor.

(Received March 1980)

1. Introduction

BASIC is the language most widely used in interactive systems. It has been designed specifically with a view to making incremental compilation easy. The language has been defined in such a way that a line of a program can be checked or compiled independently of the rest of the program. This lack of structure makes BASIC a very poor vehicle for teaching good programming technique. Programs are input on a line by line basis (one statement to a line) and the system will usually detect only simple syntax errors which are local to a line, each line of program being stored in something close to the original text form, and possibly also in machine code.

There are very few possibilities in BASIC for syntactic or semantic errors whose detection requires a more global analysis of the program. The only examples of such errors are an incomplete or wrongly nested FOR loop and use of an array or function which does not agree with its declaration (wrong number of subscripts or parameters). Recognition of such errors by a BASIC system is usually left until run-time. We thus have the undesirable possibility of the user developing a syntactically or semantically invalid program and not being informed of this until at some stage during the execution of the program, and then only if the invalid part is executed.

A good conversational system should never fail to inform the user when his program is syntactically or semantically incorrect. This aim is more difficult to achieve with a language like ALGOL 60 which permits a nested statement structure and which has sophisticated variable scoping rules. In such a language, a simple local alteration to one line of the text can result in a change to the global syntactic or semantic structure of the program. Thus a simple line by line approach as used in BASIC systems involves frequent rescanning of the whole program, if syntax and semantic errors are to be reported as soon as they appear.

Such an approach to conversation ALGOL 60 was found quite acceptable in a system developed for use in an introductory programming course for first year students (Atkinson and McGregor, 1978). However, for the more sophisticated user, these additional overheads are quite unacceptable, and an internal representation for the program is required such that a small change to the syntactic or semantic structure of the program requires only a small amount of reprocessing in order to represent the change in the internal representation of the program. Such a representation facilitates the implementation of a truly incremental compiler which performs minimal recompilation in response to commands which change the program. In addition since the linear textual structure of a program is far removed from the syntactic and semantic structure of the program, the usual form of text editor is not necessarily the best vehicle for expressing such changes.

There have been several approaches to the problem of providing a truly incremental conversational system for a structured language, the earliest being that of Lock (1965). He describes a data structure representation for an ALGOL 60 program which reflects its syntactic structure at a statement level. The editor is oriented towards altering this structure, the unit handled by the editor being the statement. Edit commands are provided for inserting, deleting and replacing groups of one or more statements. Execution efficiency is achieved by storing machine code for each statement. Similar approaches are described by Ryan *et al.* (1966), Bolliet *et al.* (1967) and Berthaud and Griffiths (1973).

In all these systems, semantic checking of the program is left until run-time, variable references being interpreted via a symbol table.

Breitbard and Wiederhold (1968) avoid the problem of incremental semantic checking in their ACME compiler for a subset of PL/I by treating all variables including procedure parameters as global, and postponing checks on the overall structure of the program until a run is requested.

Ayres and Derrenbacher (1971) describe an incremental compiler for JOVIAL in which edits are on a statement by statement basis. Their system attempts to identify the sections of the program which may need to be recompiled as a result of the alterations. Earley and Caizergues (1972) describe a similar system for VERS, an ALGOL-like language. In this case the edits are specified textually and no recompilation is done until the user indicates that no further edits are forthcoming. Both these systems are fairly liberal in their selection of statements requiring recompilation as a result of an edit. For example, Ayres and Derrenbacher recompile a whole subroutine if any statement in the subroutine is altered and Earley and Caizergues recompile all statements within the scope of a declaration if the declaration is altered.

The present paper describes a set of edit commands which specify alterations to a stored ALGOL 60 program. The internal representation for these programs is also described. This representation has been chosen so that a small alteration to the syntax or the semantics of the program results in minimal recompilation. Semantic errors as well as syntax errors are detected as soon as they occur whether during the initial input of the program or as a result of a subsequent edit to the program. The system described has been implemented in ALGOL 68-R.

2. Outline user view of the system

In order to meet the requirement that a good conversational compiler should report errors as soon as they occur, a one-pass subset of ALGOL 60 has been implemented—the defining occurrence of an identifier must precede the corresponding

CCC-0010-4620/81/0024-0222 $04.00

applied occurrences. A number of other ALGOL 60 features are omitted in the experimental system which has been implemented. These are listed in Appendix 1 and are relatively minor but for the omission of labels and the replacement of name parameter transfer by reference transfer.

The user types program text and the system compiles the program as it is supplied. At all stages during this process, further extension of the program is permitted only if the section of program already supplied is correct. As soon as an error is detected, the user is expected to correct the error, possibly by making a text edit on the unaccepted portion of the current line of text, or by editing the stored program which has already been accepted. An edit to the stored program would be necessary if, for example, a declaration had been omitted.

As new program text is input by the user, it is converted into an internal data structure which represents the syntactic and semantic structure of the program. Further details and examples of this data structure are given in Section 3. The original program text is not stored internally; there is sufficient information in the internal representation of the program for a textual version with standard layout to be generated whenever a listing is required.

Edits to a stored program (partial or complete) are made by using the structure editor. This editor differs from a conventional text editor in that the commands available are for manipulating the syntactic or semantic structure of the ALGOL 60 program rather than for changing characters in a piece of text. In order to use this editor, the user must have a very clear understanding of the structure of his program.

Individual parts of the program are identified by using a numbering system which is oriented towards syntactic units of the program rather than lines or arbitrary groups of characters as in a conventional text editor. The declaration and statement numbering system is illustrated in **Fig. 1.** This is the form in which the system will list a stored program. No distinction is made between blocks and compound statements as far as numbering is concerned so we shall use the term 'block' to imply either a block, in the ALGOL 60 sense, or a compound statement. Blocks are numbered according to the order of appearance of their **begin** symbols. The declarations in a block are numbered consecutively and the statements in a block are also numbered consecutively independently of the declarations. Substatements following a **then**, **else** or **do** are identified by the letters 'T', 'E' or 'D', respectively. Thus in the program of Fig. 1, the first declaration is identified as 10–10 and the statement following **do** is identified as 10.10ED.

Complete statements can be inserted, deleted and replaced by means of the above numbering system. Such alterations must be sensible in terms of the corresponding alteration to the syntactic structure of the program—for example, the statement after **then** in an if–then statement could not be deleted. Further facilities are available for identifying and changing units of a program at a level lower than a complete statement. Such changes within statements are still made by reference to the syntactic structure of the program.

Changes to declarations have to be made with the semantic structure of the program in mind. For example, there may be a number of reasons for changing the name of a local variable in an ALGOL program: we may want to change the name of the variable and all references to it, keeping the meaning of the program the same; we may want to change the name of the variable only in the declaration, thus making all applied occurrences of that variable refer to a more global variable of the same name; we may want only some of the applied occurrences to refer to a more global variable of the same name in which case some of the applied occurrences will need to be changed to the new name. When using the system described here, the user must specify precisely the *semantic* change he

```
*10                 begin
  10–10               integer i,j;
  10–20               boolean b;
  10.10               if i = 20 then
    10.10T
    *20                 begin
      20.10               i := 10;
      20.20               j := 12
    /20                 end
                        else
    10.10E              for i := 1, i + 1 while b
                        do
      10.10ED             b := i < 10;
  10.20               j := 13
/10                 end
```

Fig. 1 A simple ALGOL program used to illustrate statement structures

wishes to make, and only if this change is sensible will it be permitted. In addition, the system will always warn him when changes in the scopes of variables have taken place, thus eliminating the possibility of such changes occurring accidentally as can easily happen in a text-editor/compiler environment. Full details of the editing system are given in Section 4.

Finally, the system implemented permits execution of a syntactically complete program at any stage during its development. The data structure form of the program includes blocks of machine code for each syntactic unit and these blocks are generated and incrementally modified at the same time as the rest of the data structure.

Many conversational systems permit the execution of incomplete programs. They also permit the execution of a program to be interrupted (by user intervention or as a result of an execution error), edits to be made to the stored program, and execution to be resumed at the point where it left off. Such a sequence of operations can cause a program to produce results which could not have been produced by the original program or by the edited program. In the system described here, emphasis is on the development of a complete and correct ALGOL program and the system does not permit an incomplete program to be executed. Nor does it permit changes to be made to a program during execution. As pointed out by Atkinson and McGregor (1978), these restrictions do not preclude the possibility of execution tests at every stage during the gradual development of a program in accordance with the philosophy of structured programming.

3. Data structure representation of a program

In order to fulfil the aims introduced in Section 2, an internal program representation is required which permits reference to be easily made to any part of the syntactic or semantic structure of the program. This representation must also permit efficient implementation of specified changes to program structure. In this section we present such a representation for a program. The handling of machine code is deferred until Section 5.

3.1 *Syntax*

The syntactic structure of the program is represented by a data structure which corresponds to the syntax tree of the program. Each syntactic unit which can be manipulated by the editor is represented by a data cell which contains information about that syntactic unit, including pointers to the data cells for subsidiary syntactic units. This data structure is easily generated as a side effect of the syntax analysis of the program text.

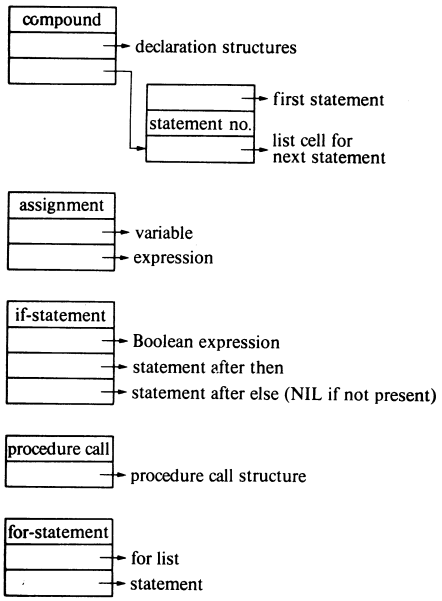

Fig. 2 Statement structures

3.1.1 *Syntax of statements.* The forms of data cell required to represent the various types of statement which can appear in a program are illustrated in **Fig. 2.** The type of any syntactic unit is represented by a 'syntax class indicator cell', unique to units of that type, which contains two procedures, one for reading and checking a syntactic unit of that type and one for generating a text listing of the syntactic unit from the information contained in its data cell.

Fig. 3 illustrates the data structure representation of the syntax of the program of Fig. 1 down to the statement level. This data structure can be very easily adjusted to take account of the insertion, deletion or removal of whole statements.

3.1.2 *Syntax of declarations.* In discussing declarations, we distinguish the *form* of the declaration (simple, array or procedure) and the *base type* of the declaration (integer or Boolean). A declaration is represented by a data cell containing a pointer to its syntax class indicator cell (which indicates the form of the declaration and contains listing and checking procedures for declarations of that form), its number, its base type and a pointer to a data structure containing information about the objects being declared.

A simple declaration list containing declarations of all three forms appears in **Fig. 4** and the syntax structure representation for these declarations is illustrated in **Fig. 5.**

3.1.3 *Syntax of expressions.* An expression is represented by a list of pointers to the data cells for the top level 'expression elements'. An expression element can be an operator, a constant, a variable, an array reference, a procedure call or a pointer to the data structure for a bracketed subexpression. Although this does not fully represent the syntactic structure of the expression as determined by the operator priorities, it provides a compromise between a full syntax structure representation and editing convenience.

3.2 *Semantics*

In order to achieve the aim of minimal recompilation, particularly with respect to changes in the semantic structure of a program, the internal form includes extensive cross referencing between declarations and the corresponding applied occurrences; it also includes information about the nested block structure of the program and hence about the scope of identifiers in the program.

Associated with each block is a 'block cell' which contains the block number, a pointer to the block cell for the smallest enclosing block and a pointer which is used to form a list of all the block cells for a program in increasing numerical order of block number. A declaration of a procedure with parameters or a function is treated as a block in order to indicate that parameters and the function designator are in scope only in the

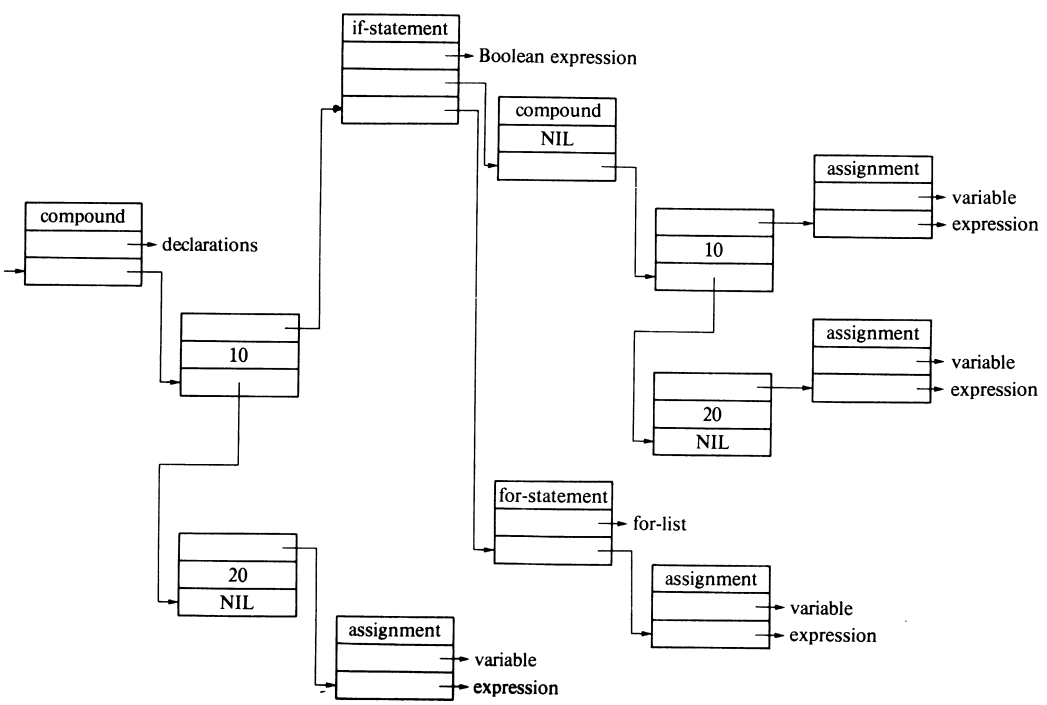

Fig. 3 Statement structure for the program of Fig. 1

```
-10      integer a, b;
-20      integer array c, d [1:2], e [1:2, 1:4];
-30      procedure f (g,h);
         value g;
         integer g, h;
         h := g;
```

Fig. 4 A simple declaration list used to illustrate the data structure representation of declaration syntax

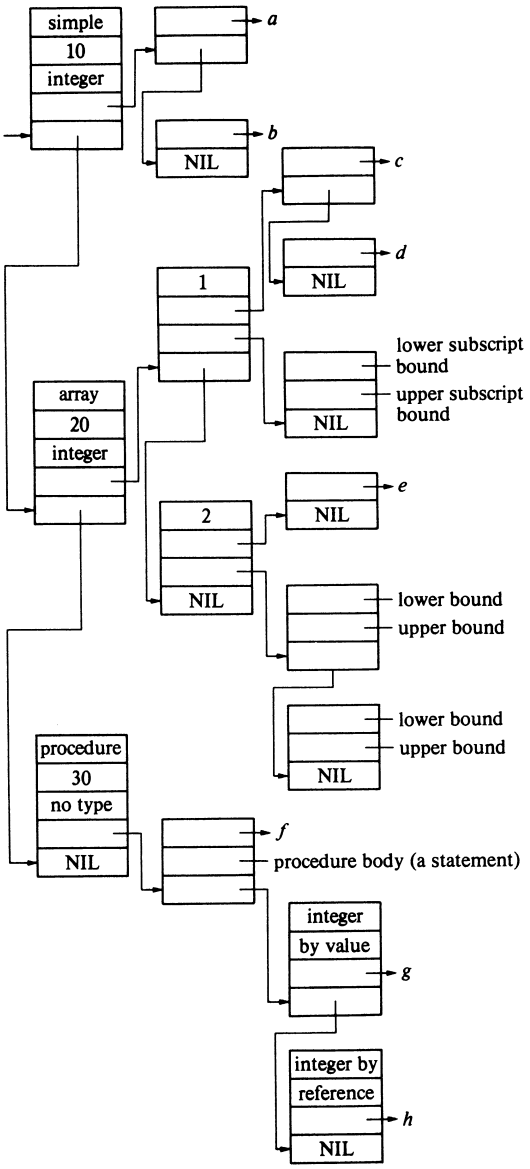

Fig. 5 Data structure representation of the syntax of the declaration list of Fig. 4

procedure declaration. Thus a block cell is associated with each such procedure declaration. The block structure representation of the simple program of **Fig. 6** is illustrated in **Fig. 7.**

Each identifier used in a program is represented by an 'identifier cell' which, in addition to the text of the identifier, contains a pointer to a data structure containing information about the scopes and applied occurrences of all objects in the program which have been declared with that name.

The main structure associated with a given identifier in the identifier cell is a list of cells, each of which contains infor-

```
*10                  begin
  10–10                integer i;
  10.10                read (i);
  10.20
  *20                  begin
    20–10                integer array a,b,c [1:i];
    20–20                procedure sum (i);
                         value i;
                         integer i;
    20–20P
    *40                  begin
      40–10                integer j;
      40.10                for j := 1 step 1 until i do
        40.10D               c[j] := a[j] + b[j]
    /40                  end;
    20–30                integer j;
    20.10                for j := 1 step 1 until i do
      20.10D
      *50                  begin
        50.10                read (a[j], b[j]);
        50.20
        *60                  begin
          60–10                integer i;
          60.10                for i := 1 step 1 until j do
            60.10D               b[j] := b[j] + a[i]
        /60                  end
      /50                  end;
    20.20                sum (i)
  /20                  end
/10                  end
```

Fig. 6 Program used to illustrate semantic structure

mation about the scope and semantics of a particular item (simple variable, array or procedure) which has been declared with that name. Each 'scope cell' in this list contains the form and base type of the corresponding item, a pointer to the block cell for the block in which the variable was declared and a pointer to an applied occurrence tree. This tree is a hierarchical representation of a list of all points in the program at which there is an applied occurrence of the item to which the tree refers. An illustration of the data structure associated with the identifier *i* in the program of Fig. 6 appears in **Fig. 8.**

Each applied occurrence of an identifier is represented by an 'applied occurrence cell' which contains a pointer to the identifier cell (and hence to information about other objects declared with the same name) and a pointer to the scope cell for the instance of that identifier to which the applied occurrence refers.

4. The structure editor

The only situation in which a conventional text edit can be performed occurs when a line containing new ALGOL program text has been typed and an error is detected during analysis of this line. The unaccepted portion of the input buffer can be changed by using simple text edit commands. All other edits must be made by using the structure editor which requires program changes to be specified in terms of changes to the structure of the stored program.

The system is initially in input mode and, until a syntactic or semantic error is detected, the user can continue input of his program. As soon as an error is detected, further input is not permitted until the program so far has been corrected. This can be done by text edits on the unaccepted section of the last input buffer or by edits on the stored partial program. At any stage, the user can interrupt input of new text to make changes to the structure of the accepted partial program. Once the final **end**

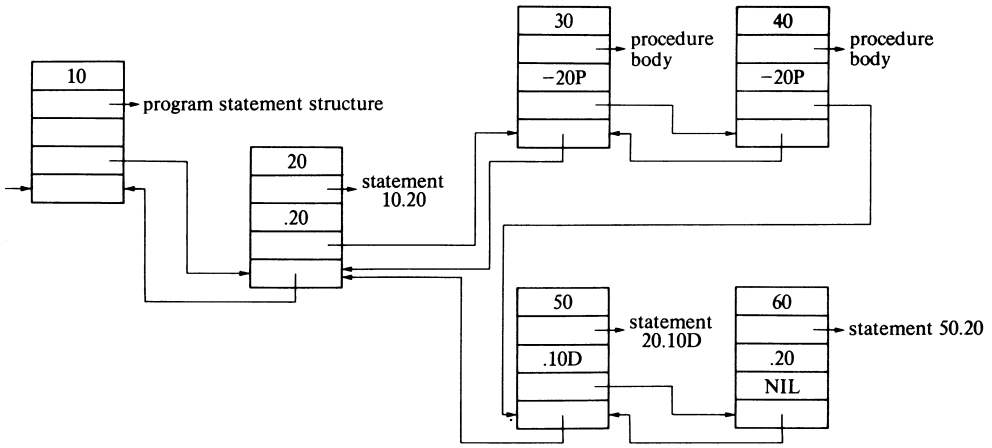

Fig. 7 Block cell structure for the program of Fig. 6

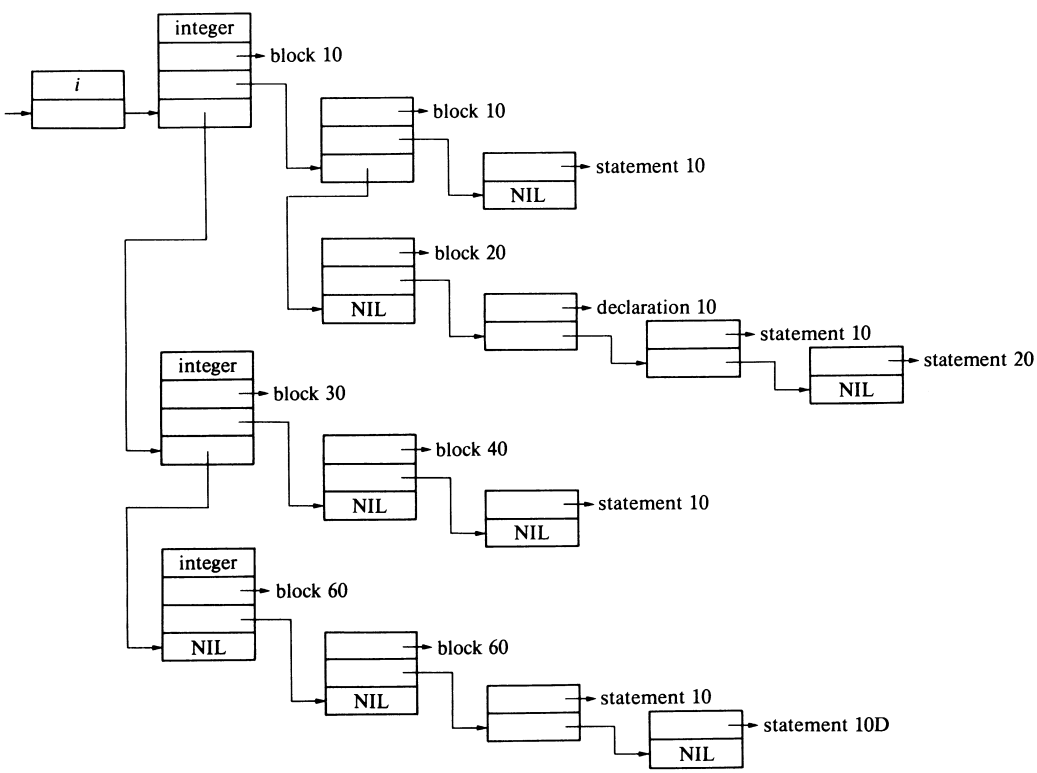

Fig. 8 Semantic structure associated with the identifier *i* in the program of Fig. 6

has been typed, the complete stored program can be manipulated only by the structure editor. This does not preclude input of extensive new sections of program text, but this can be done only when the user has identified the point in the existing structure at which the insertion is to be made.

Certain types of edit at one point in the structure may give rise to errors elsewhere in the structure which will subsequently have to be corrected. The editor is organised in such a way that alterations to the program can be made only at points before any outstanding uncorrected errors. Since we are dealing with a one-pass subset of ALGOL 60, the cause of a syntax or semantic error must be at a point in the program at or before that at which the error is reported. It should therefore always be possible to correct the earliest error in a program even with the above restriction. Whenever any edit operation is completed by the system, a message describing the earliest uncorrected error in the program is output.

A complete list of all the edit commands available appears in Appendix 2. The use of these commands is discussed in the remainder of this section. Each command can be abbreviated to the initial letters of the command words and a command is distinguished from ALGOL program text by being preceded by the command symbol '?'. All edits involving the specification of new program structures require the text for the new structure to be typed starting on the line following the edit command.

In order to specify parts of the existing program structure for the purpose of editing that structure we require the ability to identify groups of declarations or statements in the program. This enables edit commands to manipulate the program structure at the statement or declaration level and also serves as

a starting point for identifying syntactic units at a level lower than a complete statement or a complete declaration.

The program numbering system was described in Section 1 and this numbering system is used in the edit commands for the identification of single declarations or statements as well as groups of declarations or statements. Specification of such a group takes one of the following three forms where *bno* is the number of the smallest enclosing block.

(1) *bno*—(*dno*1, *dno*2) for a sequence of declarations where *dno*1 and *dno*2 are the numbers of the first and last declarations respectively.

(2) *bno*.(*sno*1, *sno*2) for a sequence of statements where *sno*1 and *sno*2 are the numbers of the first and last statements respectively.

(3) *bno*—(*dno*1, .*sno*2) for a group of declarations followed by a group of statements in the same block.

Initially, the editor has the whole program at its disposal for editing. However, for convenience when editing a deeply nested part of the program, it is possible to restrict the editor's range or 'window' to a substructure of the complete program. This simplifies subsequent identification of program elements within the window.

The window specification command *In Statement* followed by the identification of a statement within the current window makes the statement specified into the current window, thus restricting subsequent editing to substructures of the specified statement. The *In Declaration* command has a similar effect. The *In Term* and *In Variable* commands described later are used to reduce the window to a syntactic structure at a level lower than a statement or a declaration. The command symbol '?' appearing on a line by itself undoes the effect of the most recent window specification command.

The *List* command can be used to print out the current editing window together with the corresponding statement numbering.

4.1 *Statement edits*

As in the earlier systems, commands are provided for inserting, deleting or replacing any group of one or more statements. Three additional commands are available for manipulating a program at the statement level: *Statement Qualify*, *Statement Extend* and *Statement Unqualify*. *Statement Qualify* enables a group of existing statements in the program to be 'qualified' by (i.e. preceded by) **if** . . . **then**, **if** . . . **then** . . . **else**, or **for** . . . **do**. *Statement Extend* permits the addition of an else-part to an if–then statement. *Statement Unqualify* permits the removal of all but the then-branch of an if–then statement, all but the else-branch of an if–then–else statement, or all but the statement following the **do** in a for-statement.

Each of the above alterations is easily implemented by creating new data structures for any new input and by adjusting one or two pointers in the existing data structure.

Since it is the program structure being edited, many points of ALGOL syntax needed to make a textual program unambiguous are taken care of automatically by the editor. For example, if *Statement Qualify* is applied to a group of two or more statements, they will be automatically grouped into a compound statement; if an if-statement is qualified with a further **if** . . . **then**, the data structure representation of the program makes the nested if-statement unambiguous, and its structure will subsequently be indicated in a program listing by appropriate **begin** . . . **end** brackets.

The side effects which can be produced as a result of statement edits, in the sense that edits to other parts of the program are made necessary, are very limited. Edits elsewhere can become necessary only as a side effect of inserting new constructions which refer to undeclared variables.

4.2 *Edits within statements*

Variable edit commands are predominantly for use within declarations. However, the *Variable Replace* command can be applied if the current edit window consists of an assignment statement or a for-statement. The variable replaced is that on the left of the assignment or the control variable of the for-statement. The text of the new variable (simple or subscripted) has to be typed on the line following the command. If the variable in the current statement is a subscripted one, then the edit window can be reduced to consist of that variable (including its list of subscripts) by using the command *In Variable*. The subscript expressions can then be edited by using the term edit commands described below.

A number of 'term edits' are available for editing syntactic elements smaller than complete statements or declaration. The word 'term' is used to describe any complete syntactic element (apart from a statement) at a level one lower than the current edit window. A term could be an expression, a top-level expression element (see Section 3.1.3), a for–list-element, a procedure parameter, a subscript (or even a subscript bound pair if we are editing within a declaration—see later). The particular form a term takes depends on the context. Edits to the structure at a level lower than one below the current edit window require further restriction of the edit window by use of one of the window specification commands.

There is usually a very small number of top-level terms within the current edit window and it is convenient to refer to these by number: *term* 1, *term* 2 etc. For example, at the statement level, an assignment contains only one term (the expression on the right hand side), an if-statement contains one term (the Boolean expression) and a for-statement contains one or more terms (the for–list elements). At a lower level, a step–until-element contains three terms, a while-element contains two terms, an array reference contains one or more terms (the subscript expressions), a procedure call contains zero or more terms (the parameters) and an expression involving one or more operators contains at least two terms (the top level operands). A term in fact corresponds to the top level pointers in the data structure representation of the current edit window.

Clearly, the applicability of a particular edit depends on the context: *Term Swop* for example cannot be applied when the current edit window contains only one top-level term as does an assignment statement or a procedure call with only one parameter; nor can it be used to swop two terms of different types; *Term After* cannot be used when the current edit window is an expression—inclusion of additional top level operands in an expression involves the insertion of additional operators and requires the use of 'fragment edits' described below.

An additional edit command, *Name Replace*, is available for changing the array name used in an array reference, or the procedure name used in a procedure call.

Edits within expressions at the top level, other than those involving replacing or interchanging the top-level operands (terms), require the use of fragment edits. These permit the manipulation of any sequence of complete top-level elements of the expression. Edits within one of the top-level operands of an expression require the restriction of the edit window to that operand by use of the *In Term* command followed by further term or fragment edits.

4.3 *Declaration edits*

Edit commands are available for inserting deleting and replacing whole declarations in a straightforward way. These commands differ from their statement editing counterparts in that extensive side effects may be created as a result of such edits. Deletion of a declaration may make illegal any statement which previously referred to an identifier which was contained

in the declaration. On the other hand, the deletion of a declaration may result in an extension to the scope of a variable with the same name which is declared in an outer block of the program. If this outer variable has the same type as the variable whose declaration was deleted, the program will still be legal although its meaning will have changed; if it has a different type, statements referring to it may now be incorrect.

Any errors which arise as a result of a declaration edit will be reported and added to the current list of outstanding errors. Any changes in meaning will be brought to the attention of the user in the form of warning messages. Such action on the part of the editor is made possible by the extensive cross referencing, contained in the data structure representation of the program, between declarations and applied occurrences.

4.4 *Edits within declarations*

Edits can be made within a declaration only when that declaration has been selected as the current edit window (by the window specification command *In Declaration*).

All the remarks made earlier about the side effects which can be produced as a result of edits to entire declarations apply equally well when the individual components of a declaration are being manipulated. Such side effects, whether or not resulting in an illegal program, are always reported.

Of the Variable edit commands listed in Appendix 2, those for inserting, deleting or replacing variables can be used in a straightforward fashion on a simple declaration. The *Variable Mode* command can be used to change the mode of a variable in a simple declaration.

The above variable edit commands can all be used on the array identifiers in an array declaration. In addition, term and fragment edits can be used to make changes to the array bounds. When the edit window consists of the entire declaration, each subscript bound pair is treated as a term and the pairs are numbered in the order in which they appear in the declaration. One such pair can be selected as the edit window by the *In Term* command and the two subscript bounds then become terms 1 and 2. These bound expressions can then be manipulated in their entirety by using term edits, or, following a further *In Term* command, they can be edited as described earlier for expressions.

If it is required to change the subscript bound pair for an array identifier which previously shared its subscript bound pair with other variables, the *In Variable* specification command can be used to indicate that subsequent edits to the subscript bounds apply to that array alone.

The edit commands used for manipulating simple declarations can be used for performing similar operations on the formal parameter list of a procedure declaration. The text of any new parameter input must be preceded by its specification and possibly by the word **value**. In addition the *Variable Swop* command can be used to interchange two parameters. Deleting or swopping two parameters (or subscript bound pairs) will cause the system to make the corresponding alteration to actual parameters (or subscripts) in all applied occurrences of the procedure (or array), and inserting a new parameter (or subscript bound pair) results in a request for a new actual parameter (or subscript) for each applied occurrence.

The command *Name Replace* has a special effect when applied to a declaration. It can be used to change the name of a simple variable, an array or a procedure in its declaration *and* throughout its scope. In the absence of other objects declared in outer blocks and sharing the new identifier used, this edit does not affect the meaning of the program. If any other items have been declared with this new identifier, changes in the meaning of the program or even errors could be created. Such side effects are, of course, reported.

4.5 *Copy and move commands*

The commands *Copy Statement*, *Move Statement*, *Copy Declaration*, *Move Declaration* can all be used for inserting a *textual* representation of an existing program structure (statement or declaration) into the current input buffer at the point at which the command is used. Copy commands leave the existing structure unaltered while the Move commands will delete the existing structure.

The command *Copy* can be used in a similar way for copying smaller syntactic elements. This command is followed by information which identifies the element to be copied—a statement or declaration number followed by a combination of *In Term* or *In Variable* commands.

The above commands cannot be used to copy or move the data structure representation of part of a program as this could, for example, result in an applied occurrence of a variable being moved outside the variable's scope. Instead, the program text is regenerated and reanalysed in its new context.

5. Machine code representation of a program

Several alternative approaches are available for organising the execution of a program represented by the sort of data structure we have described. The simplest approach would be to write an interpreter which operates by scanning the data structure and interpreting the structures encountered. Although slow in execution, this approach has the advantage that once any changes to the data structure have been made in response to edit commands, nothing further need be done to enable execution to take place.

Another possible approach involves storing machine code for some of the lower level syntactic units of the program, and writing an interpreter which scans the upper levels of the data structure and selects which of these blocks of code need to be executed and in what order.

In our system, we have extended the second approach so that *every* syntactic unit represented in the program data structure has associated with it a block of machine code. The machine code for each unit contains, at appropriate points, subroutine jumps to the code for each subsidiary syntactic unit. Each code block will terminate with a jump instruction transferring control to the code which is to be executed next. Two examples should suffice to make the principle clear.

The block of code corresponding to an if–then–else statement will start with a subroutine jump to the code for evaluating the Boolean expression. The code for the Boolean expression will place the value of an expression on a run-time stack and then perform a return jump to the code for the parent if-statement. The if-statement code will then execute a subroutine call to the code for either the then-statement or the else-statement, depending on the Boolean value which has been placed on the stack. The code for each of these subsidiary statements will terminate with a return to the calling code of the parent if-statement.

The machine code for an ALGOL block consists simply of a subroutine call to the code for the first declaration. The code for each declaration or statement (except the last) in the block terminates with a jump to the code for the next declaration or statement in order. The code for the last statement terminates with a return to the code for the parent block.

The subroutine jump instructions to subsidiary syntactic units and the jump instructions at the end of each unit can be treated as pointers which represent the syntactic structure of the program in just the same way as do the pointers in the program data structure described in Section 3. These jump instructions are updated by the editor at the same time as the corresponding data structure pointers.

Simple variable references are all handled indirectly through a storage location associated with the variable's name. Thus the

machine code for a variable reference does not have to be changed if changes in the program structure result in a new variable of that name being in scope at that point. At run-time, the appropriate reference is inserted in the variable's identifier cell by the machine code of the variable's declaration. Any previous reference associated with the identifier is restored on exit from the block in which the variable was declared.

The approach to code generation and execution outlined in this section provides a mode of execution which is an efficient alternative to either of the two interpretive approaches mentioned. Execution of the program, once started, proceeds exclusively by means of machine code execution with no time-consuming intervention of an interpreter. By treating the branch instructions in the machine code in the same way as the pointers in the data structure, the control structure of the machine code program can be maintained during editing with little extra effort beyond that involved in generating machine code for any new ALGOL text which is typed in.

Once a program has been completely developed to the user's satisfaction, the separate machine code blocks could be very easily consolidated into a conventional machine code program.

6. Conclusions

The system described provides an effective means for the advanced programmer to develop a program without repeatedly going through the compile, execute, text-edit cycle. Use of a 'machine code data structure' in which control transfer instructions mirror the role played by pointers in the more conventional data structure representation of a program ensures reasonable run-time efficiency. Transformation of this code structure into a block of contiguous code would be a simple matter.

Programs are represented by complex data structures which impose large storage overheads but, as hardware costs decrease and paging becomes standard practice, this becomes less of a drawback. By way of compensation, the explicit representation of the program syntax tree and its accompanying semantic structure provide an ideal environment for global flow analysis and subsequent program optimisation.

As implemented, the system demonstrates the feasibility of a truly incremental compiler for a block-structured language with minimal reprocessing being performed in response to edits made to a program.

Acknowledgement

This work was supported in part by the Science Research Council.

Appendix 1 ALGOL 60 features omitted from the experimental system

the type **real**
strings
comments in procedure headings
code-bodied procedures
multiple assignments
own variables
empty statements
monadic plus
switches
goto statements
labels
call by name (replaced by call by reference)
forward references

Appendix 2 Summary of edit commands

All edit command words can be abbreviated to their initial letter.

(1) General edit commands.
Each of these commands can, in a suitable context and by use of a suitable prefix, be applied to a statement, declaration, variable, term or fragment.

After
Before
Delete
Replace

We can use, for example, *Statement Delete*, *Declaration Delete*, *Variable Delete*, *Term Delete* and *Fragment Delete*.

(2) Commands applicable only to statements.

Statement Extend
Statement Qualify
Statement Unqualify

(3) Window specification commands.

In Statement
In Declaration
In Term
In Variable

(4) Commands applicable only to variables within a declaration.

Variable Mode
Variable Swop
Name Replace

(5) Command applicable only to terms.

Term Swop

References

ATKINSON, L. V. and MCGREGOR, J. J. (1978). CONA—A conversational Algol system, *Software—Practice and Experience*, Vol. 8 No. 6, pp. 699–708.

AYRES, R. B. and DERRENBACHER, R. L. (1971). Partial recompilation, *Proceedings AFIPS* 1971 *SJCC*, pp. 497–502.

BERTHAUD, M. and GRIFFITHS, M. (1973). Incremental compilation and conversational interpretation, *Annual Review in Automatic Programming*, Vol. 7 Part 2, pp. 95–114.

BOLLIET, L., AUROUX, A. and BELLINO, J. (1967). DIAMAG: A multi-access system for on-line Algol programming, *Proceedings AFIPS* 1967 *SJCC*, pp. 547–552.

BREITBARD, G. Y. and WIEDERHOLD, G. (1968). The ACME compiler, *Proceedings IFIP Congress* 1968, pp. 358–365.

EARLEY, J. and CAIZERGUES, P. (1972). A method for incrementally compiling languages with nested statement structure, *Communications of the ACM*, Vol. 15 No. 12, pp. 1040–1044.

LOCK, K. (1965). Structuring programs for multiprogram time-sharing on-line applications, *Proceedings AFIPS* 1965 *FJCC*, pp. 457–472.

RYAN, J. L., CRANDALL, R. L. and MEDWEDEFF, M. C. (1966). A conversational system for incremental compilation and execution in a time-sharing environment, *Proceedings AFIPS* 1966 *FJCC*, pp. 1–21.