A transformation-directed compiling system

R. H. Pierce* and J. Rowell

International Computers Limited, Computer Development Division, Wenlock Way, West Gorton, Manchester M12 5DR

A translator writing system built into a high level language is described. The system enables a user to define a context free syntax for a programming language, with automatic production of tables to drive an LL(1) parser. Semantic processing is provided by means of a powerful transformation grammar from the language syntax to a standard tree representation. Practical experience with the system while implementing a complex compiler is also described.

(Received October 1975)

1. Introduction

It is some ten years since the introduction of the Compiler-Compiler (Brooker, Morris and Rohl, 1962), but during this time relatively few languages have incorporated the ideas introduced there for writing compilers. The Compiler-Compiler facilities incorporated into Atlas Autocode (Morris and Rohl, 1967) were probably the first attempt to include such compiler writing facilities into a higher level and more widely used language. While the Compiler-Compiler enjoyed considerable success, the corresponding facilities in Atlas Autocode were used only for the production of a few experimental compilers for simple languages.

The formal syntactic and semantic facilities offered in these systems have largely been abandoned in favour of faster analyser algorithms where the compiler designer can compromise between the speed of the analyser and the complexity of the semantic routines, usually at the expense of code optimality.

This was certainly the object of the Systems Program Generator (Morris and Wilson, 1970). However the lack of semantic formalism in SPG does make the design more prone to obscure error, as the authors have found during the implementation of a cross compiler from ICL 1900 series to ICL 2900 series machines. The language, STAPLE, used internally in ICL for the production of test programs and associated system software, is similar in the facilities it offers to CORAL and RTL/2, and was implemented using a 1900 SPG compiler. During its development many unplanned facilities were added which resulted in the compiler becoming slower and less reliable because the format and contents of the analysis record produced by SPG was under the total control of the compiler writer and was consequently designed around the initial language. At each enhancement stage it became more and more difficult to add extra facilities.

The opportunity arose to rewrite the 1900 based compiler for the 2900 machine and at the same time, implement enhancements which approximately doubled the language complexity. The authors decided, in the light of the SPG experience and the current state of the art in compiler writing, to go back to the original design aims of the compiler-compiler and reconsider the problems. This resulted in extensions to the compiler-compiler facilities and the invention of a formal semantic transformation grammar.

This paper describes the translator writing facilities that were used in the construction of the STAPLE compiler. These facilities were originally provided during development by a preprocessing program, but they are now part of the STAPLE language. The compiler can thus compile itself. The description of the facilities is given from the point of view of a user who wishes to use them for compiler construction. Considerations of space prevent the inclusion of any details of the actual design of the STAPLE compiler or the way in which the compiler-compiler facilities were used. It is hoped that these topics and a description of how the facilities are implemented, will be the subject of a future paper.

The compiler-compiler facilities fall into two parts.

1. A notation for defining the grammar of a programming language as a set of context free productions. This differs only in matters of style from other compiler-compiler systems.
2. A transformational grammar which maps the context free grammar onto a standard or canonical tree representation of the program. This transformation grammar is not context free, i.e. the tree produced can be made to depend on semantic information collected during the parsing of the program. Optimisation and object code production are then performed on the standard tree, which, using the transformation grammar, can be generated in a form which makes these tasks relatively straightforward.

2. Formal approaches to semantic analysis

The importance of tree structures in compiler construction has been recognised for many years, particularly with regard to the translation and optimisation of arithmetic expressions. It is well established that the compilation process can be regarded, at least theoretically, as involving the creation, transformation and flattening of trees (McClure, 1972) representing the semantic structure of programs. Increasing attention is now being given to formal methods for manipulating semantic trees, by means of transduction grammars (Lewis and Stearns, 1968) which map phrase structure grammars on to abstract trees, and transformational grammars (De Remer, 1974). The latter term is normally applied to devices which map abstract or semantic trees into other forms thereof.

The STAPLE system falls into the former category; in general outline, the parser produces a parse tree, which is then traversed, transformations being applied during traversal to produce the abstract tree representation of the program. The idea of producing a parse tree then traversing it is familiar from the early translator writing systems such as the Compiler-Compiler and the PSYCHO system of Irons (1963). However, in the present system the parse tree is intended as an aid to semantic analysis and is not a necessary consequence of the parsing method. If the situation does not warrant the production of a parse tree, transformation may optionally be performed during parsing.

The justification for using the transformation grammar in the construction of the STAPLE compiler was twofold.

1. The compiler was two-pass and some suitable way of representing the interface between the two passes was urgently required. As Currie has remarked the design of a

*Now with Software Sciences Limited, London and Manchester House, Park Street, Macclesfield, Cheshire, SK11 6SR.

two-pass compiler can be a major intellectual exercise (Currie, 1971) and any means of reducing the effort is useful.

2. The code generation and optimisation phases could be cleanly separated from the parsing and validation stages, leading to a better understanding of each. It is worth noting here that the abstract tree representation of a program can also be made highly independent of both the original language and the target machine. It is therefore an excellent candidate for a common compiler target language (Capon *et al*, 1972). Section 6 describes the authors' experience with the system.

3. The STAPLE Compiler-Compiler system

The overall design model of a compiler produced by the STAPLE system is shown in **Fig. 1,** although many individual variations are possible. In particular, the use of two passes is by no means essential.

Syntax analysis

The syntax definition facilities are quite conventional. (A BNF definition of the phrase structure facilities and the transformation grammar is given in Appendix 1). In principle, any parsing algorithm that is capable of producing a parse tree could be used. In fact, the parser chosen is of the LL(1) type, the input grammar being converted into this form by the method used in the SID program (Foster, 1968), except that left recursion is not allowed in the input. The alterations made to the grammar are invisible to the user. Note also that if the grammar cannot be made into LL(1) form, a backtracking parser can be generated. **Fig. 3** gives an example of the written form of the syntax; the notation is in fact based on the Atlas Autocode system.

Lexical analysis is either performed by user defined procedures,

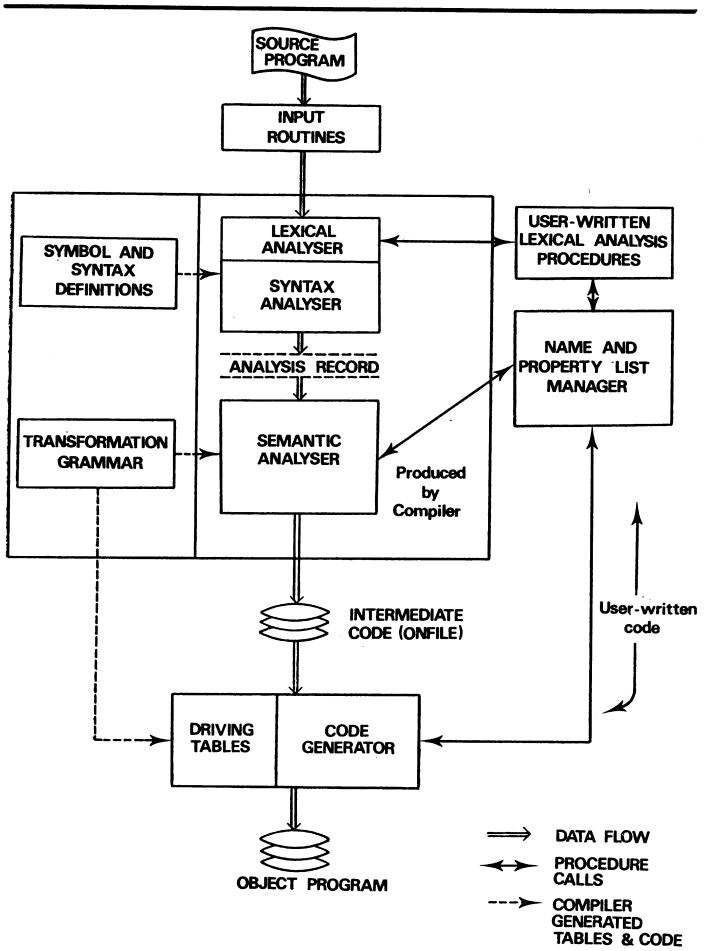

Fig. 1 Structure of compiler produced by STAPLE facilities

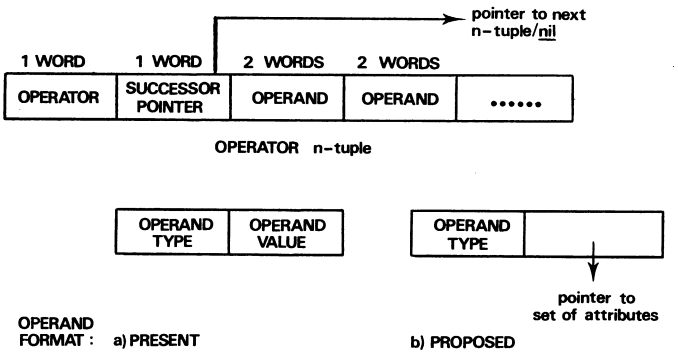

Fig. 2 Intermediate code operator format

```
FORMAT CLASS EXAMPLE
SYMBOL ":=" = 1
SYMBOL "+" = 2
SYMBOL "-" = 3
SYMBOL "*" = 4
SYMBOL "/" = 5
SYMBOL NAME = 6
SYMBOL NUMBER = 7
SYMBOL "(" = 8
SYMBOL ")" = 9
SYMBOL "INT;" = 10
SYMBOL "REAL;" = 11
SYMBOL "DEC;" = 12
FORMAT STATEMENT = ASSIGNMENT/ ....
PHRASE ASSIGNMENT = NAME, ":=", EXPR
PHRASE EXPR = MODEQUALETY, OPERAND, REST/NIL
PHRASE MODEQUALETY = MODEQUAL/NIL
PHRASE MODEQUAL = "INT; "/"REAL;"/"DEC;"
PHRASE OPERAND = NAME/NUMBER/"(",EXPR,")"
PHRASE OPERATOR = "+"/"-"/"*"/" /"
ENDCLASS
```

Fig. 3 Example of syntax definitions

or by a terminal symbol recognition procedure provided as part of the standard run time package, or both. The tables for the symbol recognition procedure are set up from the spellings of literal symbols,

e.g. SYMBOL ':=' = 29

causes an entry for ':=' to be inserted into the tables, and given the token value 29. On the other hand,

SYMBOL NAME = 36

declares that 'NAME' will be recognised by a user defined mechanism and that the lexical analyser will return 36 as the token number. The form SYMBOL (36) in a production may also be used to represent a terminal symbol.

Actions or 'phrase proceudres' may be included in the grammar, so that it would be possible to use the system as a conventional syntax directed compiler, where the actions were used to generate code directly. This is not however the intention of the system. A phrase procedure always has a Boolean result and if the procedure returns *false* a syntax error is signalled.

Several other points arise from the definition in Appendix 1.

(*a*) A FORMAT is equivalent to an ordinary production except that on recognition of a format an associated processing procedure is entered. This follows the Atlas Autocode system. Normally a format procedure will invoke the transformation system to start semantic processing. At least one FORMAT must be present to act as the sentence symbol of the grammar.

(*b*) The (CR) qualifier on a phrase indicates that no semantic significance is associated with it and that it may be ignored during transformation.

(*c*) The (TF) qualifier indicates that transformation should take place immediately recognition of the phrase is complete. If

all phrases are marked (TF) no parse tree is built, all transformations having taken place during parsing.

Semantic analysis

The semantic analyser consists of two parts—the transformations themselves, which actually produce intermediate code, and the set of supporting procedures. These procedures perform functions that cannot (yet) be expressed formally, such as adding entries to the property list, and checking the semantic validity of identifiers. The transformation system is described in Section 5 and the format of the intermediate code tree is given in Section 4.

Code generation

Having produced a tree representation of a program or statement, the rest of the compilation process must still be coded by conventional means. In the STAPLE compiler itself, the tree is traversed once to perform register allocation and some optimisation, then traversed again to produce an abstract target machine code. This final traversal may be driven by a table produced from the definition of the operator nodes in the transformation grammar. Clearly these final stages would benefit from further formalisation, by means of 'tree-to-tree' or 'tree-to-string' transformational grammars. Another area that might repay investigation is how to handle semantic information in a more systematic fashion during transformation. The object here is to reduce the amount of 'hand' coding needed to deal with this information.

4. The intermediate code format

The structure of the intermediate code (ICODE) is that of a tree, with certain features intended to aid the code generator in processing it. Each node in the tree consists of an operator, a successor pointer (possibly *nil*) and zero or more operands. Each operator has a fixed number of operands. An operand consists of a type part and a value part, as shown in **Fig. 2.**

The type part is used to distinguish the various kinds of operand (e.g. identifiers, constants of various modes, character strings or pointers to subtrees). The use of the value part depends on the operand type. For example it would hold a pointer to the property list entry for an identifier, or the value of a constant if this would fit into a single word. The ICODE tree is constructed in an array called QLIST, the root of this tree and the next free space being referred to by standard names, QLISTSTART and QPTR. This enables informal access to be made to the tree area if required.

The use of the successor pointer enables a general driving procedure to work along a chain of nodes, calling an appropriate processing procedure to deal with each operator in turn. In this way the individual operator handling procedures do not have to be concerned with their successors. Further, when a procedure is called to deal with an operator, the general driver passes to it operands of the operator as parameters (hence the fixed number of operands for a given operator). Any invocation of further procedures which will arise if a subtree is present, is always carried out indirectly by calling a 'fetch operand' procedure with the operand pointing to the subtree as its parameter. This procedure may then call the general driver again. Each operator handling procedure need thus only be aware of its own parameters and a relatively small set of global indicators and service procedures in order to plant efficient code for each operator. This greatly improves the structure of the code generator.

5. The transformation system

The basic function of the transformation system is to traverse the parse tree produced by the syntax analyser and apply the appropriate transformation at each node in this tree. The result of this process is an ICODE tree. The shape of this ICODE tree may be quite different from that of the parse tree, considerable reordering being made possible by the fact that transformations may have parameters and may contain conditional sequences. The presence of parameters allows semantic information to be passed around during transformation, and this feature gives the system considerable power. Without parameters, transformations would be very limited in their usefulness.

A simple introduction to the principles of the transformation system is given below. This does not attempt a rigorous definition of the system, the aim being to give the general flavour of the scheme. Appendix 1 gives the syntax of the transformation grammar.

The TRANSFORM statement

This is the fundamental facility of the system. Each alternative in the syntax will normally have a TRANSFORM statement associated with it. The transformations are written separately from the syntax. This is because transformations tend to be several lines long and the syntax would become difficult to read if both were written together.

A transformation statement in general produces a portion of ICODE tree. The *result* of a TRANSFORM is always an ICODE operand, which normally points to the subtree produced by that TRANSFORM. Whenever a production name appears on the righthand side of a TRANSFORM statement the corresponding TRANSFORM for the production is invoked and the operand produced is used by the current transform.

As an example consider the definitions:

```
PHRASE EXPR = TERM, '+', EXPR /
              TERM, '-', EXPR / TERM
PHRASE TERM = FACTOR, '*', TERM/
              FACTOR, '/' TERM/FACTOR
PHRASE FACTOR = NAME/NUMBER
TRANSFORM EXPR (TERM) INTO TERM                          (1)
TRANSFORM EXPR (TERM, '+', EXPR) INTO                    (2)
      PLUS (TERM, EXPR)
TRANSFORM EXPR (TERM, '-', EXPR) INTO                    (3)
      MINUS (TERM, EXPR)
TRANSFORM TERM (FACTOR, '*', TERM) INTO                  (4)
      MULTIPLY (FACTOR, TERM)
TRANSFORM TERM (FACTOR, '/', TERM) INTO                  (5)
      DIVIDE (FACTOR, TERM)
TRANSFORM TERM (FACTOR) INTO FACTOR                      (6)
TRANSFORM FACTOR (NAME) INTO NAME                        (7)
TRANSFORM FACTOR (NUMBER) INTO NUMBER                    (8)
```

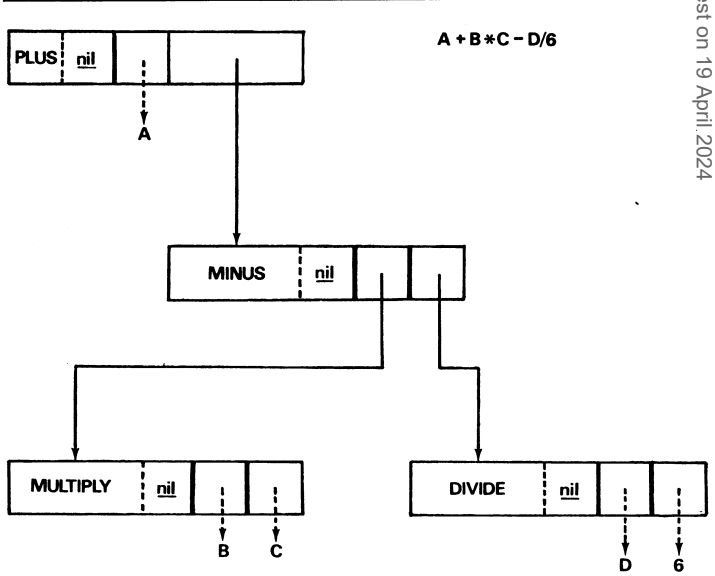

Fig. 4 Example of simple tree

where PLUS, MINUS, MULTIPLY and DIVIDE are operators each having two operands. If the input EXPR were

A + B * C − D / 6

then **Fig. 4** shows the tree that would result from the application of the transformations.

Note that transformations (1), (6) and (7) illustrate the case where the transform of a phrase does not produce any separate ICODE but merely invokes another transformation directly.

A TRANSFORM statement may also produce a terminal operand, i.e. one which forms a leaf of the ICODE tree. This is simply written, e.g.

TRANSFORM ⟨phrase⟩ INTO
OPERANDNAME VALUE ⟨expr⟩

or

TRANSFORM ⟨phrase⟩ INTO NIL

where NIL represents a null operand with a special meaning (see below).

The FOLLOWED BY construction

This construction is used to express the fact that nodes in the ICODE tree may be connected by their successor pointers, as well as in a hierarchy. List structures can thus be created. FOLLOWED BY (usually abbreviated to FB) may thus be regarded as a concatenation operator.

An example of this might be as follows.

```
PHRASE STATEMENTLIST = STATEMENT, ';',
    STATEMENTLIST/STATEMENT
PHRASE STATEMENT = ASSIGN/CALL/. . . .
PHRASE ASSIGN = NAME, ':=', EXPR
PHRASE CALL = 'CALL', NAME, PARAMETERS
PHRASE PARAMETERS = '(', PLIST, ')'
PHRASE PLIST = EXPR, ',', PLIST/EXPR
TRANSFORM STATEMENTLIST (STATEMENT, ';',
    STATEMENTLIST)
    INTO STATEMENT FB STATEMENTLIST          (9)
TRANSFORM STATEMENTLIST (STATEMENT) INTO
    STATEMENT                                (10)
TRANSFORM STATEMENT (ASSIGN) INTO ASSIGN     (11)
TRANSFORM STATEMENT (CALL) INTO CALL         (12)
TRANSFORM ASSIGN (NAME, ':=', EXPR) INTO
    ASSIGNOP (NAME, EXPR)                    (13)
TRANSFORM CALL ('CALL', NAME, PARAMETERS) INTO
    CALLOP (NAME, PARAMETERS)                (14)
TRANSFORM PARAMETERS ('(', PLIST, ')') INTO
    PLIST                                    (15)
TRANSFORM PLIST (EXPR, ',', PLIST) INTO
    PARAMOP (EXPR) FB PLIST                  (16)
TRANSFORM PLIST (EXPR) INTO PARAMOP (EXPR)   (17)
```

The result of these transformations as applied to the statements

A := B + 6; CALL P(B, Y + 1); Y := B

is given in **Fig. 5.** In transformation (9) STATEMENTLIST will in general yield a list of statement nodes, this list being concatenated on to the end of that produced by previous parts of the TRANSFORM. In this case STATEMENT only produces one node, but in general each item in a FOLLOWED BY list can produce a chain.

As mentioned above, all TRANSFORMs produce an ICODE operand. This may be the null operand NIL and this is regarded as being an empty list for the purpose of the FOLLOWED BY operator. The use of this feature is described below.

Adding parameters to transformations

As mentioned above, much of the usefulness of the system springs from the ability to pass parameters between transformations. In many ways the parameters are similar to affixes in Koster's CDL system (Koster, 1971a, b). However, in the present system the parameters are used at transformation time and do not affect the parse in any way. Transformations may also have local variables, so that each transform may be thought of as a procedure in a special notation. All transformations of a given phrase name must have the same number and type of parameters. A transformation with parameters is written

```
TRANSFORM ⟨phrase⟩ WITH (⟨value parameters⟩)
    RETURNING (⟨reference parameters⟩)
    USING (⟨local variables⟩)
```

WITH parameters are called by value in the ALGOL 60 sense while RETURNING (or RT) parameters are called by reference. All parameters and variables take the form of ICODE operands, although they may be used as ordinary integer quantities when required.

Since they are operands, however, it is possible to generate a piece of ICODE tree in one transformation and pass it to another via the parameter mechanism. This provides a method of generating a tree substantially different in structure from the parse tree.

A simple use of parameters for semantic information passing is the following. Suppose (as is often the case) the mode of

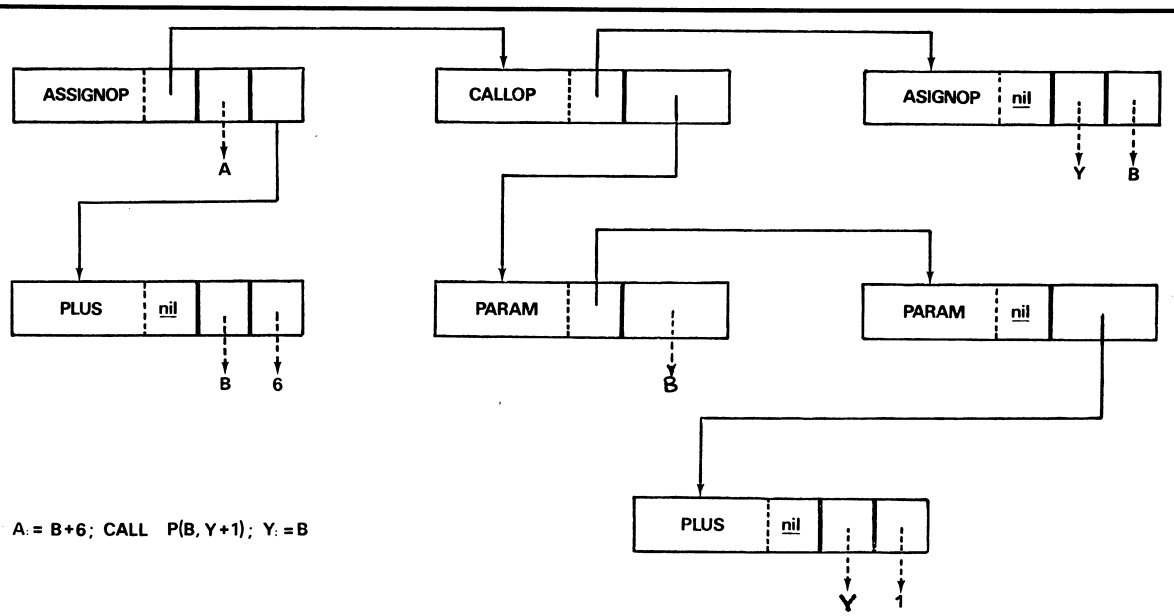

Fig. 5 Example of chained nodes

evaluation of an expression is that of the LHS of an assignment. Then transformation (13) above could be modified to read

```
TRANSFORM ASSIGN (LHS, ':=', EXPR) USING (MODE)
        INTO ASSIGNOP (LHS RT (MODE),
                       EXPR WITH (MODE))          (18)
```

The transform of 'LHS' returns the mode in MODE and this is subsequently passed to EXPR, the transforms of which would now appear as

```
TRANSFORM EXPR (TERM, '+', EXPR) WITH (MODE)
        INTO PLUS (TERM WITH (MODE),
                   EXPR WITH (MODE))              (19)
```

etc.

Note that all transform statements are executed *strictly* from left to right.

Procedures, assignments and conditionals

In the preceding section it was noted that the LHS of an assignment could return its MODE to be passed on to subsequent transforms. However the MODE must initially be derived from somewhere (normally the property list entry for an identifier). This is achieved by means of 'operator procedures' which are entirely analogous to actions in parsing. They provide an escape to informally coded parts of the compiler, and serve two purposes

(*a*) to interface to the compiler's internal tables and perform complex semantic checks

(*b*) to generate ICODE that cannot be expressed easily by TRANSFORMs alone.

Operator procedures have the same parameter specifications as transform statements and they can produce ICODE by the same means. The statement

YIELD ⟨result clause⟩

produces an operand and passes it back to the caller just as in a TRANSFORM statement.

To keep the number of operator procedures to a minimum certain other facilities are introduced. An assignment statement, e.g.

A BECOMES OPERATOR (OP1, OP2)

causes the assignment of an operand pointing to OPERATOR to the parameter or variable A. BECOMES can be used in either a TRANSFORM or an operator procedure; it always has the value NIL so that it functions as an empty list in FOLLOWED BY constructions.

Conditions of the form

IF ⟨conditions⟩ THEN ⟨result clause⟩ ELSE ⟨result clause⟩ FI

and case statements

CASE ⟨integer expression⟩ IN . ., . ., . . . ESAC

provide a most useful means of controlling the action of a transformation. The condition is usually a test on a parameter in a TRANSFORM.

Specifications

Operator and operand specifications are required, so that the compiler can perform compile time checks on the consistency of TRANSFORM statements. The possibility of run time checks is also present.

An OPSPEC statement declares

1. The operator name
2. The number and class of its operands
3. The type of operand that will point to it (up to now it has been assumed that all operands pointing to trees have the same implicit operand type, but the type must in fact be declared)
4. The name of a procedure that will be called to process it during code generation (used to set up the driving table)
5. The value that will be placed in the operator field in the node.

The last two items are optional. An example of an operator definition might be

```
OPDEF ASSIGNOP (NAMEOP, EXPRESSION)
    YIELDS SEMANTIC TYPE VALUE 29 PROCESSED BY
        APPROC
```

where NAMEOP, EXPRESSION and SEMANTICTYPE are operand types.

AN OPDEF statement has two functions:

1. To declare a terminal operand type, e.g.

   OPDEF NAMEOP (INT VALUE)

   This introduces an operand type and declares its attributes. At present the only attribute allowed is VALUE but an extension of the system provides for a number of attributes of various modes. This will enable more detailed semantic information to be represented directly in the ICODE tree.
2. To declare an operand class to be a union of other operand types and classes.

Error handling

The ERROR statement is used to signal a transformation time error. If an ON part is present a special error handling procedure is called with the parameters in the ON list, otherwise the transformations are abandoned and control returns to the procedure which first invoked the transformation system.

Miscellaneous

Certain details have been omitted from the outline of the system given above. While these details are important to a user of the system it would be tedious to describe them all in this paper. Some points are worth mentioning however.

(*a*) An actual parameter in a WITH part may be a ⟨result clause⟩ (see the syntax in Appendix 1), i.e. it looks like the RHS of a TRANSFORM. In this case the result clause is executed and the value yielded is passed. Similarly USING variables may be initialised on first entry to the TRANSFORM. In the WITH case if the parameter is a phrase name it is *not* transformed but the *untransformed* parse tree node is passed, thus allowing transformation to be deferred. To force transformation, an empty WITH list is appended to the phrase name

(*b*) The transformation of a phrase procedure (action) name has the effect of extracting an operand from the parse tree, the procedure having left it there originally. This is useful for extracting, e.g. values of constants from the parse tree

(*c*) The error handling system is more elaborate than that indicated above

(*d*) The CHAIN statement specifies which operators may be chained by their successor pointers to form a list. Operators not in a CHAIN statement may only appear in tree structure

(*e*) The YIELDS statement in a TRANSFORM has the dual function of assigning a value to its lefthand side and yielding the same value as the result of the TRANSFORM. This is useful on occasions.

6. Experience with the transformation grammar

The authors and their colleagues have used the compiler-compiler facilities described above in the implementation of the STAPLE compiler. The facilities were provided by a preprocessing program which translated the syntax and transformations into a set of initialised data areas and procedures. These were then compiled and linked into the rest of the compiler. The great benefit of the system lay in the fact that the

designer of the first pass could specify its output formally; this avoided the need to have programmers to translate his (usually ill expressed) requirements into a conventional program. The only coding required was for the syntax analyser, the phrase procedures, the operator procedures and the name and property list handler, a much less formidable task. The definition of the syntax, transformations and operator procedures took three months for one of the authors (J. Rowell). The implementation of the preprocessor required four man-months but the implementor did not have to be an expert in the language or the compiler. Skilled design effort is thus employed in the most effective manner.

Some statistics for the total number of source lines in the first pass will be of interest.

Symbol definitions	300
Syntax	181
Operator definitions	85
Operand definitions	11
Transforms	526
Operator procedures and format procedures	1,177

The figure of about 1,700 lines for the transformations and operator procedures compares very favourably with an estimate of about 10,000 lines for the semantic analysis and intermediate code generation phase of the compiler written in a conventional manner. The reduction in the size of the source text of the first pass leads to a corresponding reduction in coding errors, while the fact that the semantics are largely expressed in a formal notation has several advantages:

1. Readability
2. Directness of expression, i.e. logical errors are less liable to be introduced than in a conventional scheme where the designer's ideas must be converted to a program
3. Ease of modification and enhancement (many new language features rejected as too complex were made possible by the transformation system)
4. Extensive selfchecking facilities are provided by the transformation grammar, so that the majority of errors are detected at compile time.

In addition, a useful debugging tool was generated automatically from the transformation grammar. This would analyse the ICODE trees produced and print them in structured form with operator and operand names attached.

The overall result was that the first pass was largely correct within a few weeks of the start of testing. The vast majority of faults were in the more complex operator procedures and other informally coded parts, and these were quickly located and corrected.

Although the preprocessor introduces certain inefficiencies, the size of the first pass is not more than 40% greater than if it had been hand coded and when the transformations are *compiled* the resulting code will be smaller than the hand coded version.

The success of the formal methods used in the first pass leads to the desire to formalise the optimisation and code generation phases of the compiler. The effort expended on the second pass was 75% of the compiler, that on the first pass 25%, but each pass was comparable in complexity. At present code generation is carried out by ordinary STAPLE procedures, although the structure of the intermediate code makes the code generator well structured and reliable.

It is hoped that this paper may encourage other groups to adopt similar formal approaches to compiler construction. The theory is being developed, but the practical use of such methods is scarce, particularly in industry, where many manufacturers persist in building compilers by archaic methods. Only by trying out new approaches in 'real life' situations can the technology of language processors advance.

Acknowledgements

The authors wish to thank all their colleagues on the compiler project for their unstinting efforts. Particular thanks are due to our managers, for enabling the developments described in this paper to proceed and to Mr. A. Reiblein for implementing the preprocessors and helping generally with the transformation system.

Appendix 1

Syntax of phrase definitions

⟨syntax defn⟩ ::= ⟨phrase defn⟩|⟨format defn⟩|
⟨symbol defn⟩
⟨format defn⟩ ::= FORMAT ⟨production⟩
⟨phrase defn⟩ ::= PHRASE ⟨options⟩⟨production⟩
⟨symbol defn⟩ ::= SYMBOL ⟨literal symbol⟩ = ⟨number⟩|
SYMBOL ⟨name⟩ = ⟨number⟩
⟨production⟩ ::= ⟨NAME⟩ = ⟨alternative list⟩
⟨alternative list⟩ ::= ⟨alternative⟩⟨rest of alt list⟩
⟨rest of alt list⟩ ::= /⟨alternative⟩⟨rest of alt list⟩|⟨nil⟩
⟨alternative⟩ ::= ⟨element⟩⟨rest of alt⟩
⟨rest of alt⟩ ::= ,⟨element⟩⟨rest of alt⟩|⟨nil⟩
⟨element⟩ ::= ⟨phrase name⟩|⟨format name⟩|
⟨phrase procedure name⟩|NIL|
⟨literal symbol⟩|⟨symbol value⟩
⟨phrase name⟩ ::= ⟨name⟩
⟨format name⟩ ::= ⟨name⟩
⟨phrase procedure name⟩ ::= ⟨name⟩
⟨literal symbol⟩ ::= '⟨character string⟩'
⟨symbol value⟩ ::= SYMBOL (⟨number⟩)
⟨options⟩ ::= ⟨option⟩⟨option⟩|⟨option⟩|⟨nil⟩
⟨option⟩ ::= (CR)/(TF)

Syntax of transformation grammar

⟨tg statement⟩ ::= ⟨opspec⟩|⟨opdef⟩|⟨chain⟩|⟨transform⟩|
⟨yields⟩|⟨yield⟩|⟨becomes⟩|
⟨operator procedure⟩
⟨opspec⟩ ::= OPSPEC ⟨name⟩⟨operand option⟩ YIELDS
⟨name⟩⟨process⟩⟨opval⟩
⟨operand option⟩ ::= (⟨operand class list⟩)|⟨nil⟩
⟨operand class list⟩ ::= ⟨operand class⟩⟨restofolist⟩
⟨restofolist⟩ ::= ,⟨operand class⟩⟨restofolist⟩|⟨nil⟩
⟨operand class⟩ ::= ⟨name⟩|UNION(⟨operand class list⟩)
⟨process⟩ ::= PROCESSED BY ⟨name⟩|⟨nil⟩
⟨value⟩ ::= VALUE ⟨expr⟩|⟨nil⟩
⟨operator procedure⟩ ::= OPPROC ⟨name⟩⟨parameter sets⟩
⟨typeety⟩⟨nl⟩⟨staple program⟩
ENDOP
⟨typeety⟩ ::= YIELDS ⟨name⟩|⟨nil⟩
⟨opdef⟩ ::= OPDEF ⟨name⟩⟨attributes⟩|OPDEF ⟨name⟩
UNION (⟨operand class list⟩)
⟨attributes⟩ ::= (⟨attribute list⟩)|⟨nil⟩
⟨attribute list⟩ ::= ⟨attribute⟩⟨rest of attribute list⟩
⟨rest of attribute list⟩ ::= ,⟨attribute⟩⟨rest of attribute list⟩|
⟨nil⟩
⟨attribute⟩ ::= ⟨data type⟩⟨attribute name⟩
⟨chain⟩ ::= CHAIN ⟨operand class list⟩
⟨transform⟩ ::= TRANSFORM ⟨phrase part⟩ INTO
⟨result clause⟩
⟨yield⟩ ::= YIELD ⟨result clause⟩
⟨yields⟩ ::= ⟨name⟩ YIELDS ⟨result clause⟩
⟨becomes⟩ ::= ⟨name⟩ BECOMES ⟨result clause⟩
⟨phrase part⟩ ::= ⟨name⟩(⟨alternative⟩)⟨parameter sets⟩
⟨parameter sets⟩ ::= ⟨parameter set⟩⟨parameter sets⟩|⟨nil⟩
⟨parameter set⟩ ::= ⟨parameter type⟩(⟨namelist⟩)|USING
(⟨list of result clauses⟩)
⟨parameter type⟩ ::= WITH|RETURNING|RT
⟨result clause⟩ ::= ⟨option⟩⟨chained items⟩
⟨chained items⟩ ::= FOLLOWED BY ⟨result clause⟩|FB
⟨result clause⟩|⟨nil⟩

⟨option⟩ ::= ⟨operator⟩⟨operand part⟩|
⟨phrasename⟩⟨app⟩|
⟨operator procedure name⟩
⟨app⟩|⟨name⟩ YIELDS ⟨result clause⟩|
⟨name⟩ BECOMES ⟨result clause⟩|ERROR-
⟨number⟩⟨on part⟩
|NIL|⟨terminal operand⟩|
IF ⟨conditions⟩ THEN ⟨result clause⟩ ELSE
⟨result clause⟩
FI|CASE ⟨expression⟩ IN ⟨list of result
clauses⟩⟨outpart⟩ESAC

⟨operand part⟩ ::= (⟨list of result clauses⟩)|⟨nil⟩
⟨list of result clauses⟩ ::= ⟨result clause⟩⟨rest of res list⟩
⟨rest of res list⟩ ::= ,⟨result clause⟩⟨rest of res list⟩|⟨nil⟩
⟨app⟩ ::= ⟨parameters⟩|⟨nil⟩
⟨parameters⟩ ::= ⟨parameter⟩⟨rest⟩
⟨rest⟩ ::= ⟨parameter⟩⟨rest⟩|⟨nil⟩
⟨on part⟩ = ON (namelist)|ON ( )|⟨nil⟩
⟨terminal operand⟩ ::= ⟨name⟩⟨values⟩
⟨values⟩ ::= ⟨value⟩⟨values⟩|⟨nil⟩
⟨value⟩ ::= ⟨attribute name⟩(⟨expression⟩)
⟨outpart⟩ ::= OUT⟨result clause⟩|⟨nil⟩
⟨parameter⟩ ::= ⟨with or ret⟩(⟨list of result clauses⟩)
⟨with or ret⟩ ::= WITH|RT|RETURNING
⟨attribute name⟩ ::= ⟨name⟩
⟨phrase name⟩ ::= ⟨name⟩
⟨operator procedure name⟩ ::= ⟨name⟩
⟨staple program⟩ ::= any piece of correct STAPLE code
⟨nl⟩ ::= newline symbol
The syntax of ⟨condition⟩ and ⟨expression⟩ are part of the STAPLE language and are not described here.

References

BROOKER, R. A., MORRIS, D., MACCALLUM, I., and ROHL, J. S. (1962). The Compiler-Compiler. *Annual Review in Automatic Programming*, Vol. 3, London: Pergamon.

BROOKER, R. A., MORRIS, D., and ROHL, J. S. (1967). Compiler-Compiler Facilities in Atlas Autocode, *The Computer Journal*, Vol. 9, p. 350.

CAPON, P. C., MORRIS, D., ROHL, J. S., and WILSON, I. R. (1972). The MU5 compiler target language and autocode, *The Computer Journal*, Vol. 15, p. 109.

CURRIE, I. F. (1971). Algol 68-R in *Algol 68 Implementation*, Peck, J. E. L. (Ed), Amsterdam: North-Holland.

FOSTER, J. M. (1968). A Syntax Improving Program, *The Computer Journal*, Vol. 11, p. 31.

IRONS, E. A. (1963). *Annual Review in Automatic Programming*, Vol. 3, p. 209.

KOSTER, C. H. A. (1971a). Affix Grammars, in *Algol 68 Implementation*, Peck, J. E. L. (Ed), Amsterdam: North-Holland.

KOSTER, C. H. A. (1971b). A Compiler-Compiler, MR127, Matematisch Centrum, Amsterdam.

LEWIS, P. M., and STEARNS, R. E. (1968). Syntax-Directed Transduction, *JACM*, Vol. 15, p. 465.

MCCLURE, R. M. (1972). An appraisal of compiler technology *Proc. Spring JCC*, 1972, Vol. 40, p. 1.

MORRIS, D., WILSON, I. R., and CAPON, P. C. (1970). A System Program Generator, *The Computer Journal*, Vol. 13, p. 248.

DE REMER, F. L. (1974). Transformational Grammars, in *Compiler-Construction*: *An Advanced Course*, Bauer, F. L. *et al.* (Ed), Springer-Verlag.