A compact form of one-track syntax analyser†

By H. R. Simpson*

This paper describes a technique for automatically generating a one-track syntax analyser and discusses its use in a number of examples. A method of effecting semantic control over syntax is introduced.

(Received January 1969)

1. Introduction

J. M. Foster (1968) has described a program called SID (Syntax Improving Device) which can take in a syntax rule set containing embedded actions, automatically produce an equivalent one-track form, and then output a syntax analyser to run on a given object machine. The form of the analyser is ideally a code program which checks the syntactic legality of the input data and arranges the calling of embedded actions as and when required; it does not produce a formal parsing of the input data, although the actions could be programmed to perform this function. SID has proved to be a valuable aid in compiler writing over a wide range of practical applications, extending from simple assembly languages through to quite complicated high level languages (Currie and Griffiths, 1967).

The work described in this paper is based on a program called SAG (Syntax Analyser Generator, Simpson, 1968). SAG and SID require similar forms of input data and can produce similar forms of output analyser. However, SAG does not undertake automatic rule transformation; instead it checks the syntax for one-track properties and outputs data to assist in the process of manual transformation into an equivalent one-track set. SAG is a simpler program than SID and can be quickly and easily implemented on quite small computers with limited software facilities. Nonetheless it provides a powerful and flexible aid, particularly when writing compilers to a given specification.

Early versions of SAG produced syntax analyser programs in code which were identical in form to analysers produced by SID. However, for certain machines, this led to a rather lengthy analyser, and so an alternative interpretive form of output was developed. In this case SAG produces data which can be operated on by a simple interpreter program. The resulting analyser can be made extremely compact with very little penalty in terms of running time. This paper concerns itself entirely with the interpretive form.

The principal elements of the technique are illustrated in **Fig. 1**. Syntax rules containing actions are processed by SAG; SAG may reject the results if they do not correspond to an acceptable one-track form. If the rules are one-track, then SAG produces a data set which can be operated on by a simple interpreter program. This data set represents a close-packed form of the rules, and includes additional information to assist in the process of syntax analysis. The data set and the interpreter program together form a syntax analyser which can operate on basic symbols generated by a preprocessor. The preprocessor derives the basic symbols from the syntactic content of the input data. The analyser matches the basic symbols against the data set representing the syntax rules and generates a stream of action calls. These action calls control the sequence and timing of the actions which must be programmed to effect the required transformation between input and output. The actions may require value data associated with the input; the preprocessor derives this value data from the semantic content of the input. Also the actions may need global workspace for intercommunication purposes and the storage of intermediate results. This is a very condensed summary of the technique; the various component parts are fully discussed in the paper and a number of examples illustrate how the method might be used in practice.

This paper shows how syntax directed techniques can be applied in a direct and straightforward manner. The content of the paper covers four main areas:

(*a*) Detailed description of a simple application as it would finally appear in the object machine. This breaks down into four parts: preprocessor (Section 2), SAG output (Section 5), interpreter (Section 6), and actions (Section 7).
(*b*) Brief description of the way in which SAG produces the data for the interpreter (Sections 3, 4, 5).
(*c*) Discussion of embedded actions, and the way in which these communicate with one another via a global data structure (Sections 7, 8, 9).
(*d*) Discussion of the way in which semantics can be incorporated into the analyser (Sections 10, 11).

Sections 2–9 use examples that are quite close to those given in a paper by Foxley and King (1968) which discussed Compiler-Compiler type techniques. These examples have been chosen so that the reader can compare the Compiler-Compiler approach with the methods described in this paper.

2. The preprocessor

The input stream is passed through a 'preprocessor' before being fed to the analyser in the form of basic symbols for analysis. The preprocessor can make use of several techniques to condense the input data. It can:

* *Ministry of Technology, Royal Radar Establishment, Malvern.* Now at Bukit Gombak, RAF Tengah, Singapore

(*a*) Ignore certain characters (e.g. layout characters, blank tape, etc.).
(*b*) Combine characters in a serial fashion (e.g. reduce language words such as BEGIN, END, CODE to single symbols).
(*c*) Combine characters in a parallel fashion (e.g. treat all lower case letters as the same symbol).

The preprocessor produces two distinct outputs, shown as 'basic symbols' and 'values' in **Fig. 1**. The first of these outputs, which represents the syntactic content of the input data, is passed to the analyser in two logically equivalent forms. For each basic symbol we pass over:

(*a*) An integer representing the basic symbol. This is the basic symbol reference number (*rn*), and conventionally these numbers run from zero upwards.
(*b*) A bit pattern, all 0s except for a 1 in a position which uniquely characterises the basic symbol. This bit pattern is known as a single bit boolean word (*bw*); a boolean word may occupy one or more computer words.

Both means of representing the basic symbols are necessary since either form may be required by the analyser for the purposes of rule alternative selection and terminal symbol matching.

The value output from the preprocessor represents the semantic content of the input data. Some basic symbols may have a value associated with them (e.g. digits, letters, etc., but not usually BEGIN, END, etc.) which is required for use by the actions embedded in the syntax rules. In this case the preprocessor outputs a value (*t*0) in addition to the basic symbol described above. It is convenient if the preprocessor also remembers the value (*t*1) associated with the previous basic symbol read—this enables the actions to operate on value data associated with the terms on either side of the actions.

Table 1 gives the correspondence between input characters, values and basic symbols for Examples 1 and 2 of this paper. The words START, FINISH are examples of characters being condensed in a serial fashion, and digits 0–9 and lower case letters a–j are examples of characters being condensed in a parallel fashion. The value associated with an input character corresponds to the internal representation used in the computer. The boolean words are given in binary and each occupies a single computer word (assumed >11 bits). In more realistic practical applications many more basic symbols would be needed, and this would result in several computer words for each boolean word. The preprocessor is a fairly trivial piece of program and can be readily altered to allow new sets of language words, different groupings of input characters, etc.

In this paper a clear distinction is made between 'basic' symbols and 'terminal' symbols. Basic symbols have been adequately described above. Terminal symbols are those terms (excluding actions) in the syntax rules which are not expanded any further. Terminal symbols are fully described in the next section.

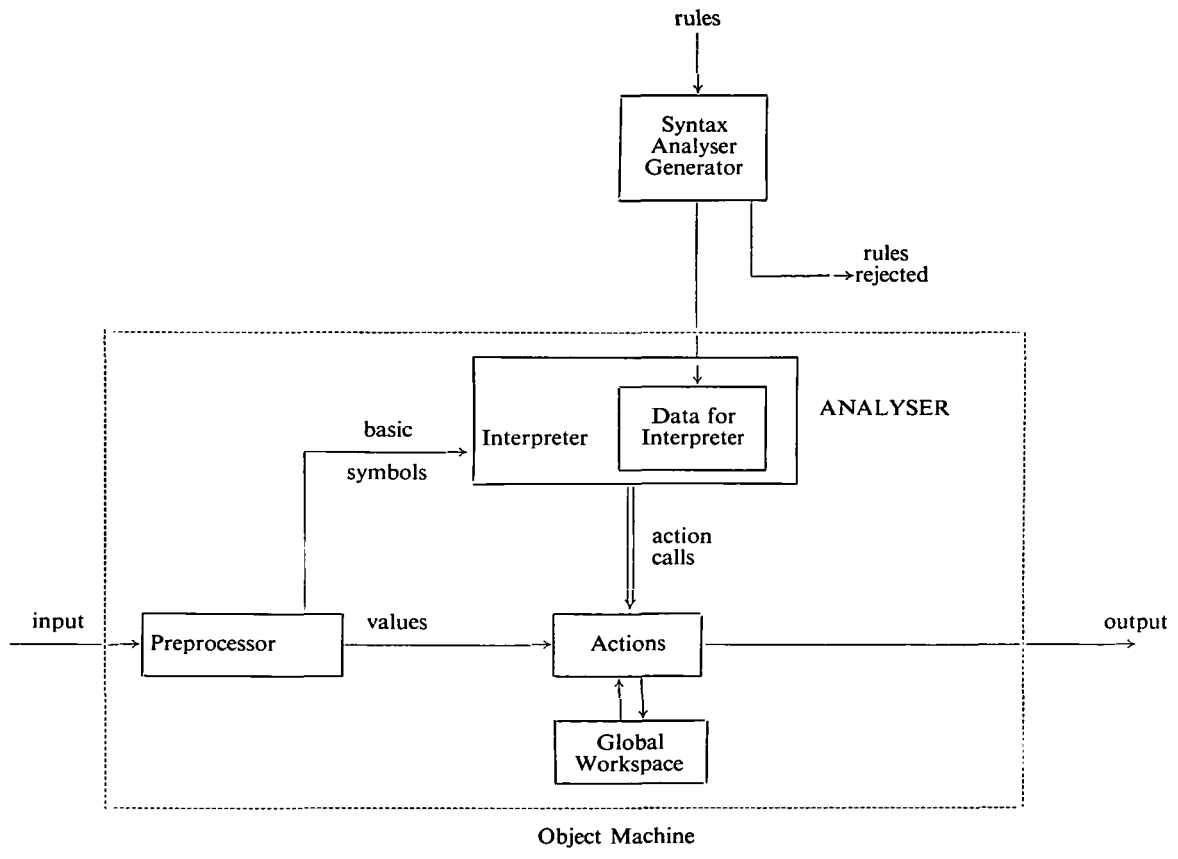

Fig. 1. Principal elements of the technique

3. SAG input, illustrated by Example 1

Three types of term appear in the syntax rule input to SAG, and a special convention for writing the identifiers of each type will be adopted to avoid the necessity for separate type declarations.

(*a*) class names—identifiers consist of italic letters and digits.
(*b*) terminal symbols—identifiers consist of upper case letters (thus all characters required in a syntax rule set must be represented by a name in upper case letters).
(*c*) actions—identifiers consist of bold lower case letters and digits.

Strictly speaking the actions are not part of the syntax rule definition, but they indicate the point at which some semantic function would need to be carried out. Each syntax rule is defined in terms of one or more alternatives. Each alternative is enclosed in round brackets and consists of either a void or a single term, or a list of terms. Individual terms in a list of terms are separated by commas. The following set of syntax rules for a restricted arithmetic expression (Foxley and King, 1968) is written using this convention.

```
rae       =(term, PLUS, rae)                    (1)
            (term)
term      =(primary, TIMES, term)
            (primary)
primary   =(LETDIG)
            (ORB, rae, CRB)
```

The terminal symbol names PLUS, TIMES, ORB, CRB, stand for the characters + * (). LETDIG is used to denote a letter or digit.

This syntax rule set would not be acceptable to SAG because it is not one-track. Also it does not specify the correspondence between terminal symbols in the syntax rules and basic symbols produced by the preprocessor. Further, the rule set is not of much practical use since it contains no embedded actions. Consider now rewriting the results in a form which

(*a*) is one-track.
(*b*) specifies the correspondence between terminal symbols and the preprocessor output.
(*c*) requires the start and finish of the input stream to be the symbols START, FINISH.
(*d*) contains actions to produce the reverse Polish form of the input.

The rules in this new form are given below and will be used as the first example in this paper. It is seen that there are two distinct parts (2*a*, 2*b*) to the rules.

Rules for Example 1

```
input     =(START, rae, stop, FINISH)           (2a)
rae        =(term, rae1)
rae1      =(PLUS, rae, punchplus)
term       =(primary, term1)
term1     =(TIMES, term, punchtimes)
            ( )
primary   =(LETDIG, outoperand)
            (ORB, rae, CRB)
```

```
LETDIG    =(0, 1)                               (2b)
START     =(2)
FINISH    =(3)
ORB       =(4)
CRB       =(5)
PLUS      =(6)
TIMES     =(7)
```

The first part of this rule set (2*a*) consists of syntax rules containing embedded actions. Each action name has been chosen to correspond with the semantic function which it performs (see Section 7 for a full definition of the actions).

The second part of the rule set (2*b*) defines terminal symbols in terms of basic symbols. Each terminal symbol (as named on the left-hand side) is equivalent to a set of one or more basic symbols (as represented by the list of reference numbers enclosed in brackets on the right-hand side). For example, LETDIG embraces all the input characters which produce basic symbols with reference numbers 0 or 1. If desired, terminal symbol definitions can overlap in the sense that they can contain common basic symbols. This method of defining terminal symbols gives useful flexibility in practical applications.

4. SAG processing

The SAG program is able to accept data punched in a form similar to that of rule set (2). SAG uses the terminal symbol definitions to set up its own internal form of the corresponding boolean words. SAG uses the syntax rules to set up a representation of the syntax in a form which gives convenient access on a rule-by-rule, alternative-by-alternative and term-by-term basis. The first syntax rule is assumed to be the starting rule for the syntax. The SAG program is fully described elsewhere (Simpson, 1968); a bare outline is given below.

The aim of the SAG program is to determine the set of all basic symbols which can be expected by the interpreter when it comes to examine the leading term of every alternative for all rules. Terms which are actions can be entirely ignored in this process. If the leading term is a class name which has a void alternative then the basic symbols for the following term must be added into the set. This process must be repeated for a given alternative until a solid term (i.e. a terminal symbol or a class name with no void alternatives) is reached, or until the end of the alternative is reached. If the end of the alternative is reached by this process, then we must include the basic symbols for the leading terms of all rules which can follow the rule in which the given alternative occurs. The process can be regarded as the computation of the set of symbols which can be seen by looking at, or if necessary through, each alternative. Boolean words are used to express the various sets of basic symbols which are relevant to the computation.

SAG includes checks on the syntax rule set. It detects the various forms of cyclic conditions which can exist, e.g.:

(*a*) single cycle
$a = (a, \ldots)$

(*b*) multiple cycle
$a = (b, \ldots)$
$b = (a, \ldots)$

(*c*) masked cycle

$a = (b, a, \ldots)$
$b = (\,)$
$(\ldots)$

It detects non one-track conditions, e.g.:

$a = (b, \ldots)$
$(B, \ldots)$
$b = (B, \ldots)$

If the syntax is completely non-cyclic and one-track then SAG will go on to produce the data to be operated on by an interpreter; otherwise it indicates where the trouble lies and manual adjustments must be made to transform the rules into an equivalent one-track set. The form of the final SAG output can vary to suit particular applications; an example is given in Section 5.

The initial input of a syntax rule set into SAG should be in a form which reduces the number of cyclic and non one-track rules to a bare minimum. Much trouble is saved if a pair of rules is always used to express the syntax of a list of items:

list = (*item*, *list*1)
*list*1 = (*item*, *list*1)
( )

Actions can be inserted at various positions in this pair of rules:

list = (**act0**, *item*, **act1**, *list*1, **act6**)
*list*1 = (**act2**, *item*, **act3**, *list*1, **act5**)
(**act4**)

These actions are called at the following times:

act0 before first item in list
act1 after first item in list
act2 before every item except first
act3 after every item except first
act4 after last item (but before act5)
act5 called $n - 1$ times at end of list, where n is the number of items in the list
act6 after last item (and after act5)

When all actions required in a particular application have been inserted, it may be possible to simplify the syntax by substituting the expansion for '*list*' in the first alternative of '*list*1'. Syntax rule set (2) shows substitutions of this nature.

The procedure outlined in the previous paragraph will usually prevent the occurrence of any cyclic rules, but rules which are non one-track may still exist. These are brought to light by a 'clash' between the symbols (represented by boolean words) associated with different alternatives in a rule, i.e. there are common legal starters for more than one alternative in the rule. Non one-track rules can be removed by a process of substitution and rearrangement (Foster, 1968).

5. SAG output, illustrated by Example 1

When SAG has checked that the syntax rules are one-track, it outputs data specifying the contents of two arrays, '*boolarray*' and '*rulearray*'. The *rulearray* data controls the selection of alternatives and gives the list of terms forming all alternative expansions; it also specifies which input basic symbols are allowed to match against a given syntax terminal symbol. The *boolarray* data is used to specify multiple bit boolean words required to perform 'one against many' checks with the single bit boolean words coming from the preprocessor. This section describes the contents of *boolarray* and *rulearray*, using Example 1 as an illustration. The reason for setting up the data in the form described can be understood by examining the interpreter program (Section 6).

The *boolarray* data produced by SAG for Example 1 is given in **Table 2**. The boolean words in *boolarray* are formed by taking the 'logical or' of single bit boolean words for basic symbols (**Table 1**), and as such represent sets of basic symbols. Thus word 0 in **Table 2** denotes

Table 1

Correspondence between input characters and basic symbols for Examples 1 and 2

INPUT	OUTPUT FROM PREPROCESSOR		
	VALUE (*t*0)	REFERENCE NUMBER (*rn*)	BOOLEAN WORD (*bw*)
0	0		
1	1		
2	2		
3	3		
4	4	0	100000 000000...
5	5		
6	6		
7	7		
8	8		
9	9		
a	32		
b	33		
c	34		
d	35		
e	36	1	010000 000000...
f	37		
g	38		
h	39		
i	40		
j	41		
START		2	001000 000000...
FINISH		3	000100 000000...
(	18	4	000010 000000...
)	19	5	000001 000000...
+	12	6	000000 100000...
*	24	7	000000 010000...
−	11	8	000000 001000...
/	15	9	000000 000100...
x	55	10	000000 000010...

basic symbol number 0 or 1 (input characters 0–9 or a–j), and word 3 in **Table 2** denotes basic symbol number 3 or 5 or 6 (input word FINISH or input character')' or '+'). Most practical applications will require more than one computer word in *boolarray* for each boolean word.

The individual words in *rulearray* are partitioned into four fields $f0, f1, f2, f3$; $f0$ and $f1$ are single bit fields, but the size of $f2$ and $f3$ can vary to suit individual applications.

Field $f2$ in *rulearray* is used to specify a basic symbol set (*bss*); *bss* can have two alternative forms. Where the basic symbol set consists of a single symbol, then *bss* is a basic symbol reference number (*rn* in **Table 1**). Where the basic symbol set consists of more than one symbol, then *bss* is the index (*bx*) for a multiple bit boolean word in *boolarray*. The two forms for *bss* are distinguished by using a marker bit in $f2$. The data in $f2$ is used by procedure *boolmatch* (see Section 6). SAG can compute appropriate values of *bss* to output in *rulearray* from previously assembled boolean word data.

There is one word in *rulearray* for each terminal symbol definition in the SAG input data. In this case the various fields have the following meanings:

$f0$—set to 1 (denotes that this is data for a terminal symbol).
$f1$—not used.
$f2$—*bss* (specifies basic symbols corresponding to the given terminal symbol).
$f3$—fault number.

There is a block of words in *rulearray* for each syntax rule. The first word in this block has $f0$ set to 0 (to discriminate between syntax rule and terminal symbol data). The block divides into two parts: data for the selection of alternatives and data specifying the alternative expansions. The fields in the alternative selection are used as follows:

$f0$—set to 0.
$f1$—set to 1 if the alternative consists of a void or a single class name or terminal symbol.
$f2$—*bss* (specifies basic symbols which are legal starters for the alternative).
$f3$—if the alternative is a void then $f3$ is set to zero; if the alternative is a single class name or terminal symbol then $f3$ gives the appropriate entry point in *rulearray*; otherwise $f3$ locates (using a pointer *pa*) the data for the expansion of the alternative in *rulearray*.

Table 2

SAG boolarray output for Example 1

INDEX (*bx*)	WORD CONTENTS
0	110000 000000...
1	110010 000000...
2	000101 000000...
3	000101 100000...

There is one word of alternative selection data for each alternative. These words are consecutive in *rulearray* and are terminated by a word in which $f0$ is set to one and a fault number is packed into $f3$. The fields in the alternative expansion data are used as follows:

Table 3

Description of SAG output for rule set (2)

ASSOCIATED NAME	DATA FOR INTERPRETER				
	$f0$	$f1$	$f2$	$f3$	
	0	1	—	*input*	
input	0	0	*rn* = 2	*pa*	——┐
	1	—	—	*faultno*	┐
	1	1	—	START	←┘
	1	1	—	*rae*	
	1	0	—	**stop**	
	0	1	—	FINISH	
rae	0	0	*bx* = 1	*pa*	——┐
	1	—	—	*faultno*	┐
	1	1	—	*term*	←┘
	0	1	—	*rae*1	
*rae*1	0	0	*rn* = 6	*pa*	——┐
	0	1	*bx* = 2	0	┐
	1	—	—	*faultno*	┐
	1	1	—	PLUS	←┘
	1	1	—	*rae*	
	0	0	—	**punchplus**	
term	0	0	*bx* = 1	*pa*	——┐
	1	—	—	*faultno*	┐
	1	1	—	*primary*	←┘
	0	1	—	*term*1	
*term*1	0	0	*rn* = 7	*pa*	——┐
	0	1	*bx* = 3	0	┐
	1	—		*faultno*	┐
	1	1		TIMES	←┘
	1	1		*term*	
	0	0		**punchtimes**	
primary	0	0	*bx* = 0	*pa*	——┐
	0	0	*rn* = 4	*pa*	——┐ ┐
	1	—	—	*faultno*	┐ ┐
	1	1	—	LETDIG	←┘ ┐
	0	0	—	**outoperand**	┐
	1	1	—	ORB	←┘
	1	1	—	*rae*	
	0	1	—	CRB	
LETDIG	1	—	*bx* = 0	*faultno*	
START	1	—	*rn* = 2	*faultno*	
FINISH	1	—	*rn* = 3	*faultno*	
ORB	1	—	*rn* = 4	*faultno*	
CRB	1	—	*rn* = 5	*faultno*	
PLUS	1	—	*rn* = 6	*faultno*	
TIMES	1	—	*rn* = 7	*faultno*	

*f*0—set to 1 if term is not the last term.
*f*1—set to 1 if term is a class name or terminal symbol
*f*2—not used.
*f*3—switch element number if term is in action (see Section 7); otherwise *f*3 is an entry point in *rulearray*.

It is clear that there is a close correspondence between the *rulearray* data and the SAG input rule set. **Table 3** shows a breakdown of the *rulearray* data for Example 1. Names appearing in the *f*3 column must be replaced by either switch element numbers (actions) or *rulearray* entry points (class names or terminal symbols). Arrows down the right-hand side locate expansion data for alternatives.

Word 0 in *rulearray* is used to determine the start-up conditions for the analysis, and it arranges the calling of the first syntax rule. Its effect can be understood by studying the interpreter program.

SAG generates the *rulearray* and *boolarray* data in a fairly straightforward manner and this data gives all the information required to carry out a syntax analysis. The data emerges from SAG in a purely numeric form. *rulearray* and *boolarray* can either be embodied in the object machine program in the form of constants, or be read in as data at the start of an analysis. The way in which *rulearray* and *boolarray* are used by the interpreter is described below.

6. The interpreter

The interpreter takes in data from the preprocessor and performs the analysis in conjunction with the syntax specification contained in *rulearray*. The interpreter program is quite short and is given below; first we need to define the various variables and procedures used in the program:

*t*1	value associated with last basic symbol read.
*t*0, *rn*, *bw*	see **Table 1**.
preprocess	a procedure which takes in the input stream and sets up values of *t*0, *rn*, *bw* as given in **Table 1**.
stack	a one-dimensional array used for remembering re-entry points in *rulearray*.
q	points to next available word in stack.
p	entry point in *rulearray*.
d	contents of a word in *rulearray*.
*f*0, *f*1, *f*2, *f*3	names of the four fields subdividing *d*.
boolmatch	a boolean procedure which unpacks the value of *bss* contained in *d* and has the value **true** if data specified by *bss* matches either *rn* or *bw* (whichever is appropriate).
fault	a procedure which unpacks and prints a fault number from *d*; normally this procedure would allow the analysis to continue from a suitable point after skipping part of the input stream; actions concerned with outputting data would usually be inhibited after a fault.
action	switch name (see Section 7).

The complete interpreter program, written in an ALGOL-like form, is as follows:

```
startup:
    t0 ← 0;
    stack [0] ← 0;
    q ← 1;
read:
    t1 ← t0;
    preprocess;
interpreter:
    q ← q − 1;
    p ← stack [q];
a1: d ← rulearray [p];
    if f0 = 0 then go to a2;
    stack [q] ← p + 1;
    q ← q + 1;
a2: p ← f3;
    if f1 = 0 then go to action [p];
a3: d ← rulearray [p];
    if f0 = 1 then go to a6;
a4: if boolmatch then go to a5;
    p ← p + 1;
    d ← rulearray [p];
    if f0 = 1 then go to fault else go to a4;
a5: p ← f3;
    if p = 0 then go to interpreter;
    if f1 = 1 then go to a3 else go to a1;
a6: if boolmatch then go to read else go to fault;
```

The stack is used to store the *rulearray* re-entry points within the expansions for alternatives. No check on stack overflow is included above, but would be needed to give a completely safe interpreter. The analysis process is initiated by going to the label '*startup*', and continues until it is stopped by an action (see Section 7).

In practice, the above interpreter program would normally be written in code to give maximum efficiency. Also action calls would be performed by inserting the addresses of the actions into *rulearray*, rather than by using a subsidiary switch.

7. Actions for Example 1

The switch declaration for the action calls required in Example 1 would have the following form:

switch *action*←*punchplus, punchtimes, outoperand, stop*.

The actions are programmed as follows:

```
punchplus:
    punch (12);
    go to interpreter;
punchtimes:
    punch (24);
    go to interpreter;
outoperand:
    punch (t1);
    go to interpreter;
stop:
    go to finish;
```

The procedure *punch*(*i*) has the effect of punching out the character which has internal representation *i* using the correspondence given in the first two columns of **Table 1**. All actions except the last return control to

the interpreter on completion. The last action terminates the program.

The complete program in the object machine has the following main components:

(*a*) Declarations of procedures, variables, arrays, switch required in (*b*)–(*e*) below.
(*b*) A preprocessor to effect the transformation specified by **Table 1**.
(*c*) *rulearray* and *boolword* data as specified by **Tables 2 and 3** (produced by SAG).
(*d*) The interpreter (Section 6).
(*e*) The actions (defined above).

The program is extremely short for this particular example. In general the length of the program depends directly on the complexity of the problem in hand, and the interpreter is the only program overhead which is problem independent.

A typical input to this program might be:

START $a + b*(c + d*e)*f$ FINISH

This produces the following output:

$$abcde* + f** +$$

It is quite instructive to perform a manual analysis of this input data using the interpreter program and data described previously.

8. Variations on Example 1

This section outlines three simple variations to Example 1 (as defined by rule set (2)). These illustrate the importance of the positioning of an action, the ease with which minor additions to the syntax can be made, and the use of global variables for handling intermediate results. Firstly consider the effect of the **punchplus** and **punchtimes** actions as they appear in rule set (2). In their present position they produce a 'right associative' form of reverse Polish; i.e.

$$a + b + c$$

is treated as:

$$a + (b + c)$$

and the reverse Polish form is:

$$abc++$$

To achieve a 'left associative' form of reverse Polish, the rules *rae*1, *term*1 in rule set(2) must be replaced by the following:

*rae*1 =(PLUS, *term*, **punchplus**, *rae*1) (3)
( )
*term*1 =(TIMES, *primary*, **punchtimes**, *term*1)
( )

In this case

$$a + b + c$$

is treated as:

$$(a + b) + c$$

and the reverse Polish form is

$$ab + c +$$

The left or right associative property of the syntax can be expressed by writing a single non one-track or cyclic rule. The right associative form of *rae* is:

rae =(*term*, PLUS, *rae*, **punchplus**) (4)
(*term*)

The left associative form of *rae* is:

rae =(*rae*, PLUS, *term*, **punchplus**) (5)
(*term*)

Rules (4) and (5) clearly indicate the associative properties of the syntax, and would form an acceptable input to a program like SID (Foster, 1968). The one-track form of rules required by SAG does not express the associative properties of the syntax particularly well, but it is nonetheless possible to see that rule set (2) will delay the output of an operator for as long as possible, whereas rules (3) will output the operator as soon as possible.

For the second variation on Example 1, consider introducing subtraction and division to have equal binding with addition and multiplication respectively. The most straightforward way of doing this is to replace the rules *rae*1, *term*1 in rule set (2) by the following:

*rae*1 =(PLUS, *rae*, **punchplus**) (6*a*)
(MINUS, *rae*, **punchminus**)
( )
*term*1 =(TIMES, *term*, **punchtimes**)
(SLASH, *term*, **punchslash**)
( )

and add two new terminal symbols to rule set (2):

MINUS =(9) (6*b*)
SLASH =(10)

Changes of this type are very easily introduced.

None of the examples considered so far has required the use of global variables for the purposes of intercommunication between actions. For the final variation on Example 1 consider an alternative way of handling the problem posed by syntax rule set (2) as modified by (6*a*, *b*) above. Rules *input*, *rae*1, *term*1 in rule set (2) are replaced by the following:

input =(START, **setup**, *rae*, **stop**, FINISH) (7*a*)
*rae*1 =(PLUSMINUS, **stkoperator**, *rae*, **outoperator**)
( )
*term*1 =(SLASHTIMES, **stkoperator**, *term*, **outoperator**)

Terminal symbols PLUS, MINUS in rule set (2) are replaced by two new definitions:

PLUSMINUS =(7, 9) (7*b*)
SLASHTIMES =(8, 10)

The actions **punchplus, punchtimes, punchminus, punchslash** are replaced by **setup, stkoperator, outoperator** which are defined as follows:

```
setup:
    i ← 0;
    go to interpreter;
stkoperator:
    s[i] ← t1;
    i ← i + 1;
    go to interpreter;
outoperator:
    i ← i − 1;
    punch (s[i]);
    go to interpreter;
```

These three actions communicate with one another via the push-down stack *s* which is controlled by the pointer *i*. This final variation on Example 1 shows the syntax can be simplified at the expense of introducing more complicated actions. In practice there is always some choice to be made as to the distribution of work between syntax and actions.

9. Example 2

The second example which will be discussed in detail is that of algebraic differentiation (Foxley and King, 1968). This turns out to be a substantially more complicated problem than Example 1. Efficient communication between actions is only achieved if a global data structure containing lists is used. Furthermore, since the basic technique only caters for the memory of a single input symbol, the actions must be programmed to remember those parts of the input stream which will be required for later use.

The SAG form of the syntax rule set for this problem is as follows:

input =(START, **setup,** *termlist*,**stop,** FINISH) (8*a*)
termlist =(*term*, **out,** *termlist*1)
*termlist*1 =(PLUS, **punchplus,** *termlist*) ( )
rae =(*term*, *rae*1)
*rae*1 =(PLUS, **push,** *rae*, **pull, formrae**) ( )
term =(*primary*, *term*1)
*term*1 =(TIMES, **push,** *term*, **pull, formterm**)
primary =(INDEPTVBLE, **formprim1**) (DEPTVBLE, **formprim2**) (CONSTANT, **formprim3**) (ORB, *rae*, CRB, **formprim4**)

CONSTANT =(1) (8*b*)
DEPTVBLE =(2)
START =(3)
FINISH =(4)
ORB =(5)
CRB =(6)
PLUS =(7)
TIMES =(8)
INDEPTVBLE =(11)

The actions are given in an abbreviated form below. Before describing them certain global variables, procedures, special conventions, etc., must be defined:

i	stack pointer.
*s*0, *s*1	push down stacks of list pointers.
*z*0	a list used to build up list of input characters.
*z*1	a list used to build up derivative of *z*0.
*w*0, *w*1	lists used for temporary storage of top stacks *s*0, *s*1.
outlist(1)	output the list 1.
c(1)	form copy of list 1.
{*l*, *m*, . . .}	form a single list of the items *l*, *m* . . . (which may be lists, characters or the variable *t*1) by joining lists end to end, and inserting the internal representation of characters or the value of *t*1 as indicated by their position.

```
setup:
    i ← 0;
    go to interpreter;
push:
    s0 [i] ← z0;
    s1 [i] ← z1;
    i ← i + 1;
    go to interpreter;
pull:
    i ← i − 1;
    w0 ← s0 [i];
    w1 ← s1 [i];
    s0 [i] ← s1 [i] ← nil;
    go to interpreter;
out:
    outlist (z1);
    go to interpreter;
punchplus:
    punch (12);
    go to interpreter;
formrae:
    z0 ← {w0, +, z0};
    z1 ← {w1, +, z1};
    go to interpreter;
formterm:
    z0 ← {c(w0), *, c(z0)};
    z1 ← {(, w0, *, z1, +, w1, *, z0, )};
    go to interpreter;
formprim1:
    z0 ← {t1};
    z1 ← {t};
    go to interpreter;
formprim2:
    z0 ← {t1};
    z1 ← {D, t1};
    go to interpreter;
formprim3:
    z0 ← {t1};
    z1 ← {0};
    go to interpreter;
formprim4:
    z0 ← {(, z0, )};
    z1 ← {(, z1, )};
    go to interpreter;
stop:
    go to finish;
```

This example has been programmed using the RRE list pack. In its present form it is apt to produce a lengthy and not particularly readable output which contains superfluous brackets, terms multiplied by zero, etc. This problem can be solved by carrying type data assocated with every list. Actions would be made conditional depending on the type of the operand lists and insertion of brackets would be delayed for as long as possible. Alternatively, the selector actions and semantically controlled syntax rules described in Section 10 could be used.

10. Context dependence

Consider the syntax for a very simple form of assignment statement:

assign = (*id*, BECOMES, *id*) (9)

If *id* can be of two types, real or int, then a compiler for this rule would have to include an action to perform any necessary type changing. This action would be positioned as follows:

assign = (*id*, BECOMES, *id*, **typechange**) (10)

The particular function performed by **typechange** will depend on the types of the two *id* terms, and this is a semantic choice. It will now be shown how such a semantic decision can be expressed within the syntax rule set.

Two new types of term are introduced into the syntax rule set:

(*a*) Selector Actions. These are actions which compute and stack the data which will later be interpreted to effect a semantic decision. Identifiers for these actions will be as for normal actions but enclosed in primes.

(*b*) Semantically Controlled Syntax Rules. These are rules in which the choice of alternative does not depend on the basic symbol produced by the preprocessor, but is decided by the data stacked by a previous selector action. Identifiers for these rules will be as for normal rules but enclosed in primes.

The data stacked by a selector action is in fact an alternative number in a semantically controlled syntax rule.

Syntax rule (10) can now be rewritten using these new types of term:

assign =(*id*, **'realint'**, BECOMES, *id*, **'realint'** **'typechange'**) (11)
'*typechange*' =('*rhsreal*')
('*rhsint*')
'*rhsreal*' =( )
(realtoint)
'*rhsint*' =**(inttoreal)**
( )

In this form of the roles, the only conditional statements occur in **'realint'**. This action would be called in the normal way by the interpreter, and would be of the following form:

realint:
if *typeid* = *unset* **then go to** *idfault;*
altno ← **if** *typeid* = *real* **then** 1 **else** 2;
go to *interpreterx;*

typeid is a procedure which is able to examine the type of the last identifier read. The action exits to *interpreterx* which stacks *altno*. When the interpreter comes across a semantically controlled syntax, it unstacks *altno* and uses it to select the corresponding alternative. This type of facility introduces considerable additional flexibility into the syntax rule set.

In many cases the semantically controlled syntax rule will be the more natural way of implementing the decision-making process which is carried out during the analysis of the input data. Consider the following syntax in which *xxx* is a class allowing several alternative expansions:

xxx =(*id*) (12)
(*id*, *parpart*)
(*id*, OSB, *exlist*, CSB)
(*id*, OSB, *ex*, CSB)
(*id*)
parpart =(ORB, *parlist*, CRB)
( )
exlist =(*ex*, *exlist*)
(*ex*)

The definition for *xxx* could occur in the syntax for a high level language in which the five alternative expansions for *xxx* are used for the following purposes:

alternative 1—real or integer
2—procedure call
3—array element
4—switch element
5—label

Syntax rule (12) gives rise to awkward clashes between the various alternative expansions. The syntax cannot determine whether an identifier on its own is a real or integer (alternative 1), a procedure call with no parameters (alternative 2), or a label (alternative 5). There is also a clash between alternatives 3 and 4. The syntax rule can be rewritten to enable one-tracking to take place, but this would complicate the functions of the actions which must be inserted in the rules. A more straightforward solution is obtained if we use a 'selector action' together with a 'semantically controlled syntax rule':

xxx =(*id*, **'idtype'**, '*xxx*1') (13)
'*xxx*1' =( )
(*parpart*)
(OSB, *exlist*, CSB)
(OSB, *ex*, CSB)
( )

It is seen that context dependence has been introduced since different routes through the syntax can now be made to depend on the semantics of the input data. The practical implementation of this is easily arranged using a program of the SAG type.

11. Implementation of context dependence

Surprisingly few changes need be made in order to incorporate context dependence. Both the SAG program and the interpreter must be slightly extended to cope with the two new types of term. The selector actions are called by the interpreter in precisely the same way as the normal actions. However, the semantically controlled syntax rules must be specially marked in the SAG output to enable the interpreter to replace the normal selection process based on the input symbol. These rules are marked by making $f0 = 1$ and $f1 = 1$ in the first word of the data for the rule; a fault number for the rule is also packed into the first word. The remainder of the data for the semantically controlled syntax rule is identical to the data for a normal syntax rule, except that no word is required to mark the end of the available alternatives. To illustrate this point, **Table 4** describes the SAG output which would be produced for syntax rule set (11) (BECOMES has been included for completeness). **Table 4** uses the same conventions as **Table 3.**

The changes to the interpreter can be discussed in terms of additions to the interpreter program given in Section 6. A stack is required to remember the values of *altno* computed by the selector actions. This stack can be conveniently located at the other end of the stack used for *rulearray* re-entry points. If *qmax* + 1

is the size of this stack and '*r*' is the stack pointer for the selector actions, then the following statement must be added to the start up sequence:

r ← *qmax;*

In order to take special action for a semantically controlled syntax rule, the following statement must be added at label *a*6 in the interpreter:

if *f*1 = 1 **then go to** *a*7;

At label *a*7 the following piece of program must be included:

```
a7: r ← r + 1;
    p ← p + stack [r];
    d1 ← d;
    d ← rulearray [p];
    if boolmatch then go to a5;
    d ← d1;
    go to fault;
```

All selector actions exit to *interpreterx* which stacks *altno:*

```
interpreterx:
    stack [r] ← altno;
    r ← r − 1;
    go to interpreter;
```

To be completely safe, checks should be included to ensure that the data at each end of stack does not meet in the middle.

Table 4

Description of SAG output for rule set (11)

ASSOCIATED NAME	DATA FOR INTERPRETER				
	*f*0	*f*1	*f*2	*f*3	
assign	0	0	*bss*	*pa*	┐
	1	—	—	*faultno*	│
	1	1	—	*id*	←
	1	0	—	**'realint'**	
	1	1	—	BECOMES	
	1	1	—	*id*	
	1	0	—	**'realint'**	
	0	1	—	*'typechange'*	
'typechange'	1	1	—	*faultno*	
	—	1	*bss*	*'rhsreal'*	
	—	1	*bss*	*'rhsint'*	
'rhsreal'	1	1	—	*faultno*	
	—	1	*bss*	0	
	—	0	*bss*	*pa*	┐
	0	0	—	**realtoint**	←
'rhsint'	1	1	—	*faultno*	
	—	0	*bss*	*pa*	┐
	—	1	*bss*	0	│
	0	0	—	**inttoreal**	←
BECOMES	1	0	*bss*	*faultno*	

Although the selection of an alternative in a semantically controlled syntax rule is done by predetermined semantic data, a syntactic check is still performed to ensure that the incoming basic symbol is a legal starter for that alternative. The values of *bss* for this purpose are computed using the algorithm for the normal syntax rules.

The syntax checking part of the SAG program must be slightly modified. Cycles should not be permitted in semantically controlled syntax rules, but of course such rules can have alternatives which clash. The banning of cycles in semantically controlled syntax rules does not prevent the use of recursive definitions in the usual manner, but merely ensures that at least one basic symbol is processed between successive calls of the rule. In order to make the technique as straightforward as possible, each selector action should only be allowed to plant a single value of *altno* in the stack. If this restriction is made then it is possible to make the SAG program scan the rules to determine the semantically controlled syntax rules which make use of data stacked by a given instance of a selector action. This would be a very valuable aid when dealing with a large complicated syntax.

12. Discussion

The SAG technique described in this paper gives a reasonably straightforward method of making direct use of formal syntax description in programs concerned with the interpretation of input data which has a complex structure. The SAG program itself can be implemented quite quickly and is suitable for quite small computers; the program requires no special software (e.g. list processing, programming language allowing recursion). SAG is a highly segmented program, and it is estimated that the size of the largest segment can be kept below 1K; the overall size of the program can certainly be kept below 8K. SAG could be usefully implemented on machines with stores as small as 8K by using 1K for program (and overlaying different segments of program in this space) and 7K for data; this amount of data space is sufficient to handle quite complicated syntaxes, e.g. Coral 66 (Currie and Griffiths, 1967). The precise forms of the interpreter program and the SAG output data will depend on the machine and the maximum size of the syntax set which is to be handled. For ease of explanation, a number of minor changes have been made to SAG as previously described (Simpson, 1968). Also SAG has not been modified to include extensions to allow the incorporation of semantics, but this is a simple matter.

It has been found that the technique whereby actions are embedded in the syntax rules imposes a very useful discipline on a program writer. The actions break the problem down into small manageable parts, and the syntax analyser is used to govern the overall flow of the program. Although examples in Sections 2–9 were of a mathematical nature, the main application of this approach lies in the construction of compilers and assemblers. The technique has proved useful even when applied to simple assembly languages which would otherwise be handled by *ad hoc* methods.

Some comparison can be made with Compiler-Compiler techniques. The SAG (or SID) approach would

appear to throw a far greater load onto the programmer; the complete syntax analysis is not available at any point, special precautions must be taken to remember past data and subsidiary results, and the actions have no parameters. However, in practice these disadvantages turn out to be relatively unimportant, particularly if the actions are able to communicate with one another via a complex data structure (e.g. lists). On the other hand the technique results in an efficient end product which can be tailored to the problem in hand.

The semantically controlled syntax rules introduced in Sections 10, 11 are thought to be a particularly important addition to the SAG approach. They allow a very flexible interaction between syntax and semantics. Particular parts of the syntax can be selected on the basis of semantics. This allows tight syntactic definitions to be applied in particular circumstances; if no semantics are used then the syntax always has to cater for the general case, and some degree of checking and control is lost. In fact the semantically controlled rules turn out to be complementary to the normal syntax rules, since many awkward syntactic clashes arise from using syntactically similar structures for semantically different categories of data.

13. Acknowledgements

The author is extremely grateful for the help received by way of informal discussion with his colleagues in the Mathematics Division, R.R.E. In particular, thanks are due to P. M. Woodward and D. P. Jenkins who have made useful comments on the first draft, and to P. R. Wetherall who has supervised the final stages of production of the paper.

References

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

FOXLEY, E., and KING, P. (1968). The Implementation of Syntax Analysis Using ALGOL, and some Mathematical Applications, *The Computer Journal*, Vol. 10, p. 325.

CURRIE, I. F., and GRIFFITHS, M. (1967). *Coral 66 Manual*, R.R.E. Technical Note No. 732.

SIMPSON, H. R. (1968). *SAG—A Syntax Analyser Generator*, R.R.E. Technical Note No. 739.

Book Review

Computers in Humanistic Research. Edited by E. A. Bowles, 1967; 264 pages. (Prentice-Hall, Inc., £3 0s. 0d.)

Some of the papers and discussions at a series of IBM-supported conferences on the role of the computer in humanistic research are here printed in somewhat abridged form. The purpose of the exercise seems to have been to assuage computer-phobia among those humanists who weren't already hopeless cases; but the diagnosis is hasty and superficial, and I doubt that the therapy adopted can have produced many lasting cures.

The sugar-coating, to begin with, is laid on suspiciously thick: 'To have done this with a desk calculator would have taken one man 105 years working constantly day and night without any breaks whatsoever. When performed on an IBM 7094 computer, the task required only 55 minutes at a total cost of about $350!'

His suspicions roused by that 'whatsoever', the patient begins to entertain systematic doubts, and soon finds it easy to believe the opposite of what he's told: 'A welcome by-product is the computer's complete dependability. . . . One ideal area for computers is inquiring into the way problems are solved. . . . The computer is frequently able to bring to light hitherto unsuspected relationships or meanings. . . . The necessity of providing an extremely lucid explanation of *what* he hopes to achieve compels the scholar to face the questions *why* he wants to do it.'

Nor is it good tactics to blame resistance on 'unreasoning abhorrence' of the computer, 'suspicion, fear, and ignorance . . . an almost prehistoric mentality'. The patient finds himself reversing roles and diagnosing in his turn that the inability to make out a fair case for the opponent must itself indicate some deep and all-too-well-founded unconfidence.

Two proposals for large-scale computerised multivariate analysis—of archaeological records in one case and historical in the other—report meeting resistance barely explicable on rational grounds. But these projects were presumably open, like any others, to objections on any of a number of grounds—their feasibility, cost-effectiveness, statistical clarity, conceptual novelty, and so on. To give the impression that opposition to computerised projects can only be attributable to machine-phobia is to betray just the kind of epistemological naïveté best calculated to nourish that phobia.

This naïveté also takes the form of fantasies of exhaustivity. Several contributors talk of collecting 'all relevant material, . . . every bit of data, . . . every measurable feature'; of 'standardising the concepts' and establishing 'extremely detailed' 'universal codes', with 'provision for infinite subdivision' of 'every descriptive variable'. Symptomatically, one author mentions with what I take to be pride only lightly touched with guilty embarrassment—like the father of a 15-pound baby—his 'sheets of matrix which, when glued together, extended for over 10 ft.' It seems urgent to repeat that something more recalcitrant stands in the way of exhaustivity than mere 'practical limitations' of channel-capacity, whether of the machine or its user.

Continued on page 250