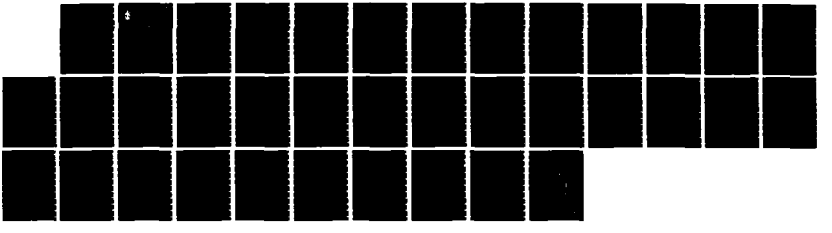
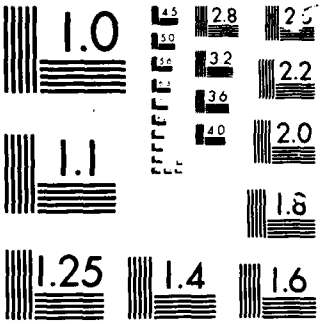
AD-A168 087 A GUIDE TO SID FOR USERS OF THE FLEX COMPUTER(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) S J GOODENOUGH ET AL. NOV 85 RSRE-MEMO-3768 1/1

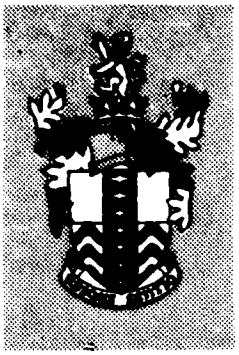
UNCLASSIFIED DRIC-BR-99097 F/G 9/2 NL





MICROCOPY CHART

UNLIMITED



RSRE MEMORANDUM No. 3768

ROYAL SIGNALS & RADAR ESTABLISHMENT

A GUIDE TO SID FOR USERS OF THE FLEX COMPUTER

Authors: S J Goodenough, P D Taylor
and G D Whitaker

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

RSRE MEMORANDUM No. 3768

UNLIMITED

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3768

Title : A Guide to SID for Users of the FLEX Computer.

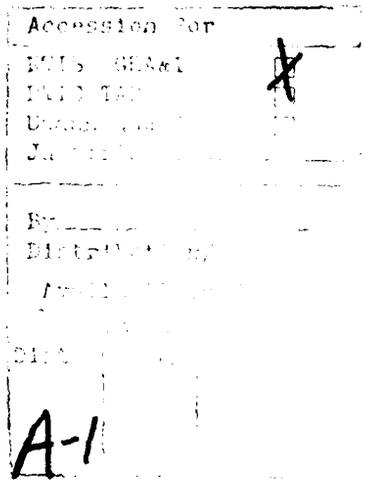
Authors : S.J. Goodenough, P.D. Taylor & G.D. Whitaker.

Date : November 1985.

Summary

This document describes a utility known as "SID" that is available on the RSRE Flex Computer. SID accepts a syntax with embedded actions, transforms it, and produces an Algol68 RS module (suitable for any RS system) that contains an analyser procedure. This analyser will check whether an input conforms to the given syntax and, while doing so, will initiate calls of the embedded actions which take the form of user-defined procedures that operate on stacks managed by the analyser. By combining the analyser module with other user-defined modules to form a complete program, syntax-directed utilities such as compilers, interpreters and translators may be constructed; indeed SID was written using itself.

Contents



1 Introduction

This document describes a utility known as "SID" that is available on the RSRE Flex Computer [3, 4, 5]. SID accepts a syntax with embedded actions, transforms it and produces an Algol68 RS [2] module (suitable for any RS system) that contains an analyser procedure. This analyser will check whether an input conforms to the given syntax and, while doing so, will initiate calls of the embedded actions which take the form of user-defined procedures that operate on stacks managed by the analyser. By combining the analyser module with other user-defined modules to form a complete program, syntax-directed utilities such as compilers, interpreters and translators may be constructed; indeed SID was written using itself.

The syntax transformations (fully described in [1]), ensure that the syntax is unambiguous and LL(1) (i.e. only one symbol need be read to determine which alternative is to be followed at any stage in the syntax).

The analyser module keeps one procedure "analyser" which takes two procedural parameters: "reader" and "syntax_error". Successive calls of the first procedure must deliver successive lexical entities from the file containing the input. The second procedure is called by the analyser whenever a syntax error is detected. The result of a call of the analyser is the result of the user-defined procedure "return" (see sections 3 & 4).

Section 2 describes the form of syntax that is acceptable to SID, and section 3 describes how to use the function "runsid" which invokes SID. The various methods of error recovery and how they may be employed by the analyser are discussed in section 4.

Section 5 describes the use of SID by means of a simple worked example and it is is recommended that a newcomer to SID should read the example concurrently with the rest of this document. The example syntax is used to construct an interpreter that behaves like a pocket calculator.

It is assumed throughout that the reader has some familiarity with Algol68 RS and Flex.

2 Preparing the Syntax

The form of the syntax required by SID is similar but not identical to Backus-Naur Form (BNF). Terminal symbols (or basic symbols - those which are not expanded in the syntax) are declared at the beginning of the syntax in the **basics** section and non-terminal symbols (or rules) are defined in the subsequent **rules** section. Note that basic symbols and rules are only distinguishable by virtue of the fact that the basic symbols are declared first in the **basics** section.

Each basic symbol is characterised by an integer (or a set of integers), given alongside the symbol in the **basics** section. These characteristic integers are used by the analyser to represent their respective symbols; this will be described further in section 3.

Rules are terminated by semicolons, and alternatives in rules are separated by commas instead of the vertical bar (|) used in BNF. An empty alternative is denoted by the dollar symbol ($) or just by nothing followed by a comma or semicolon, rather than by a Φ. Comments are written between two hash (#) characters. Names may include the visible space (_) character.

Actions can be embedded in the syntax and these appear as names inside angled brackets - not to be confused with non-terminal symbols in standard BNF. These actions allow user-defined procedures to be called during the syntax analysis and thus enable the analyser to do rather more than merely check for syntactic correctness.

For a full description of the form of the input to SID, see Appendix C.

The names of the embedded actions contain the name of a user procedure and information on the mode of its parameters (if any) and its result. The special form of the embedded action can be described more formally as:

```
action      = "<" action_name ">";

action_name = procedure_name parameters result;

parameters  = $,
              parameters parameter;

parameter   = "-" pop_or_top mode_name,
              "-lv",
              "-" small_unsigned_integer,
              "-mon";

pop_or_top  = "p",                          # p = pop #
              "q";                          # q = top #

result      = $,
              "--" mode_name;
```

Note that characters in quotes stand for themselves and that no space characters may appear between the angled brackets.

A "mode_name" should be a conventional Algol68 mode name and be at most 11 characters long. The modes BOOL and CHAR should not be used alone, although they may appear inside modes that are constructed from vectors, arrays, structures or unions. A stack is declared in the analyser module for each different "mode_name" in the "actions", and wherever "mode_name" occurs in the syntax it denotes the corresponding stack.

A "procedure_name" should be a conventional Algol68 procedure name, except that spaces are not allowed. Furthermore, in order to avoid a clash of names between action procedures and internal variables of the analyser, "procedure_name" must *not* begin with the characters "sid_".

When an action is invoked, a procedure ("procedure_name") is called and its parameters are obtained from the appropriate stacks by "pop"ping or "top"ping (see below) and the (non-VOID) result of the procedure is pushed onto the appropriate stack. A blank "result" part indicates a VOID result, and thus no stack is pushed to. The prefixes "p" and "q" indicate whether a stack is to be either "pop"ped (the top element is removed) or "top"ped (the stack is unchanged, but a copy of the top element is used). All of the procedure calling and stack management is performed by the analyser.

Other sorts of parameter may also be supplied to the action procedure. An "lv" parameter is of mode LEXVAL (see section 3) and will be the lexical value of the symbol just read. For example, if the string "42" was read then "lv" would contain the number forty-two.

A small unsigned integer parameter may be supplied as a literal integer. It must lie in the range 0 to 255 and may be used to indicate, for instance, which of several permissible terminal symbols was actually encountered, or to identify exactly what part of the syntax is currently being invoked. As a demonstration, the worked example in section 5 uses the numbers 1, 2, 3 and 4 to indicate the operators +, -, *, / (respectively).

A "mon" parameter is a value of mode MONITOR. If this form of parameter is used anywhere the mode MONITOR must be declared by the user and be available to the analyser module. MONITOR values are used to store positions in the syntax and are employed subsequently in recovering from any syntax errors (see section 4 for more details).

When a parameter takes one of the last three forms (i.e. lv, small_unsigned_integer, and mon) there is no stack involved. The analyser will supply the required information from (respectively) the representation of the current lexical item, the small literal integer

given in the action syntax, and positions held in its internal data structures.

There should be an action procedure called "return" if - as is usual - the analysis is to terminate. There are no restrictions on the parameters of "return", but its result must be a value of mode RESULT. This result value is not stacked - hence an empty result must be indicated in the syntax - but is instead returned to, and used as the result of, the analyser procedure (see section 3).

SID will check that actions are used consistently, i.e. that the modes are the same in all calls, but the form need not be identical as sometimes "p" parameters could be used and sometimes "q". At some stage (see section 3), all the procedures and modes whose names appear in "actions" must be declared by the user and the modes of the procedures must correspond to that implied by the "action" name.

An embedded action is called after the basic symbol to its right is read. Depending on the circumstances, this symbol may or may not have been checked. For example:

```
rule = basic1 <action> basic2;
```

When the procedure "action" is called, the symbol to its right will have been read, but not necessarily checked by the analyser. Hence actions in general should not assume that the symbol to the right is correct. However in situations like the following:

```
rule = <action1> basic1,
       <action2> basic2;
```

the symbol to the right will have been checked as the analyser will have needed to find out which action is to be called.

3 Running SID

The function "runsid" takes an Edfile which contains a syntax enclosed by curly brackets ("{", "}"). It reads in the syntax, attempts to transform it by calling SID and then outputs a syntax analyser based on the procedure and stack philosophy described in sections 1 & 2. One form of successful output is simply text (an Edfile), and another is the result of compiling such text (a Compiledpair).

The function will prompt the user to choose between three versions of SID:

Version A produces an analyser procedure which will only work on Flex: it is quick to compile but relatively slow to run.

Version B is a generalised version of the first and apart from minor details of the separate compilation system, its output should work on any Algol68 RS implementation. This form of output is slow to compile, but relatively quick to run. This version is recommended for Flex users when a 'final' syntax has been determined.

Version C simply checks that the input syntax is transformable by SID.

The user is also prompted with the following questions:

Question	Default Answer
SID to compile its output	yes
a print out of the original & intermediate rules	no
a print out of the cyclic replacements and final rules	no

The meaning of each of these questions will be explained in due course.

If errors are detected during the syntax reading phase, the editor is invoked and messages are sited at appropriate places in the Edfile. The result of such an edit will be the result of "runsid". Error recovery is poor; normally one error is enough to terminate the reading phase. If the syntax is read successfully but SID is unable to transform it, an Edfile is created and all messages and relevant information are put into it. The result of "runsid" is then the structure:

(original_Edfile, error_information_Edfile)

Occasionally SID will fail to transform a syntax and will give up after

a certain number of new rules have been created during the transformation. In these cases it is useful to run SID with a request to print out the intermediate rules. Examination of this output will usually reveal a rule which SID has failed to transform after repeatedly expanding it in various ways. This frequently indicates an original rule which has a number of alternatives which at some depth in the syntax have equivalent start sequences but differ in a way which cannot be resolved by one symbol look-ahead.

If "runsid" is to produce a compiled output then definitions of all the action procedures, modes of stack elements and those of LEX, LEXVAL and RESULT must be provided by the user. If version A is to be used, then the INT lexval_size is also required (this is the size, in Words, of a value of mode LEXVAL). If any "mon" parameters are used, the mode MONITOR should also be declared.

The modes LEX, LEXVAL, RESULT and MONITOR are subject to the following restrictions:

1) A value of mode LEX is used to describe the current basic symbol. The mode must be a structure which includes two fields with modes and selectors as follows:

   INT type - must be the (or one of the) characteristic value(s) of the basic symbol as given in the input to SID.

   LEXVAL val - this will be passed as the LEXVAL to those actions with "-lv" parameters. Typically it is a UNION(REF VECTOR[]CHAR, INT, REAL, ...).

2) The mode MONITOR must be a structure of two integers (see section 4).

3) The mode of RESULT should not be VOID or start with UNION.

The user supplies the required information by prefixing the "{" in the input by a sequence of Module-values that between them keep all the required objects and modes. (N.B. It will not be possible to produce a compiled output if no Modules are supplied.)

```
input = sequence-of-module-values
        "{" syntax "}";
```

The result of a successful call of runsid is a Compiledpair (or Edfile when not compiling) augmented, when requested, by an Edfile containing the listings of the syntax rules. A textual result may be merged with the necessary modules and compiled at a later date. An output (or text of an output) will look like:

```
output = analyser-module-heading
            sequence-of-module-values
               "{" syntax "}"
                  edfile1
                     sid-code
                        edfile2;
```

It is expected that when a Compiledpair is produced it will be kept in a Module, and because the input syntax (including the curly brackets etc.) is part of the text of this Module, there is no need to keep the syntax separately. If at a later time it is required to change the syntax, the text obtained from the Module may be edited; this whole text (including the syntax, sid_code, analyser, etc.) may then be re-submitted to "runsid". However, care should be taken that the syntax is still contained within the first pair of curly brackets so that the input is of the form:

```
input = analyser-module-heading
            sequence-of-module-values
               "{" new-syntax "}"
                  edfile1
                     old-sid-code
                        edfile2;
```

When "runsid" is re-run on a modified output only the "sid_code" will be replaced; the remainder will be unchanged. This has the consequence that if it is required to use one of the other versions of SID then a new input should be produced containing just the required modules and the syntax as if one were starting afresh. Failure to do this would result in a new "sid_code" with an old (incompatible) analyser procedure.

A Compiledpair delivered from "runsid" KEEPs a procedure, "analyser", and a mode INTERNALS. The mode of "analyser" is:

```
PROC (PROC LEX {reader},
      PROC(INTERNALS) BOOL {syntax_error}
     ) UNION (VOID, RESULT)
```

The parameters of the analyser procedure are supplied by the user. The first is a lexical analyser or "reader", which on successive calls should deliver a LEX value representing each basic symbol in turn from the input. The second procedure,"syntax_error", is called by the analyser whenever a syntax error is discovered.

When called, this second parameter will either attempt to recover from a syntax error and allow the analyser to continue or stop the analysis altogether. The parameter of "syntax_error" is a value of mode INTERNALS which is a structure comprised of the various internal values of the analyser. This INTERNALS value enables all the information known by the analyser to be examined and also allows some things to be altered. If "syntax_error" delivers TRUE, the analyser will continue with the analysis (presumably with certain internal variables changed),

otherwise the analyser procedure will terminate with a VOID result. For more information on INTERNALS and syntax error recovery see section 4.

Syntax analysis terminates normally when the action "return" is called. In this case, the result of the analyser is the RESULT value delivered by "return".

The internal form of the analyser produced by version A differs slightly from that of the analyser produced by the general version B. Because of this, the "syntax_error" procedure may have to be tailored to the form of analyser used (see the section on error recovery for more details). There is no other observable difference between the two forms of analyser and, with the above proviso, they may be interchanged freely. For details see the description of the SID output in Appendices A & B.

4 Error Recovery

Whenever a syntax error is discovered in the input the analyser procedure calls one of its parameters ("syntax_error"). The procedure supplied may perform syntax error recovery with as much sophistication as desired. If no error recovery is to be attempted then the procedure can simply deliver the boolean FALSE. However, even in this case, it is probably desirable for some form of error message to be output. On Flex, the editor is normally called to display the error message at an appropriate position in the data.

The mode of "syntax_error" is PROC (INTERNALS) BOOL where:

```
MODE INTERNALS = STRUCT(INT test_index {the start of the test or
                                        cascade of tests where the
                                        syntax error was found},

                        VECTOR [] CHAR sid_code {the "code" interpreted
                                                 by the analyser
                                                 procedure},

                        REF INT index {the current position in the
                                       syntax - may be changed during
                                       error recovery},

                               stind {the index of the top of
                                      sidstack - may be changed during
                                      error recovery},

                        REF LEX lex {the current lex (i.e. the symbol
                                     to the right) - may be changed
                                     during error recovery
                                    },

                        REF REF VECTOR [] INT sidstack {sid's own
                                                        internal stack},

                        VECTOR [] BOOL blwds {the boolean words used for
                                              tests where multiple
                                              choices are possible},

                        INT sid_mult {the size of each boolean word}
                       )
```

The fields of an INTERNALS value represent the current state of the data that the analyser was processing when the syntax error occurred. All of these fields may be inspected and, depending on their mode, some of them may be reset.

Effective error recovery is, in general, an intricate process. In a sense, it involves "guessing" the best way of correcting an incorrect input. This can sometimes be achieved by noticing that the current symbol is superfluous (ie. the following symbol would fit in the current position) or that a symbol is missing (ie. if only one symbol is acceptable at the current position and the current symbol is a valid continuation after that). In situations like these it is reasonable to ignore (or invent) the symbol and continue the analysis. This can be achieved within "syntax_error" by setting the "lex" field of its

parameter to a symbol that would be acceptable, and setting the "index" field to be the start of the current test (this value is given by the "test_index" field).

A procedure to determine from the "sid_code" the acceptable symbols at some stage of syntax analysis may be constructed along the same lines as the analyser procedure produced by SID. However this procedure must not call any actions; it should accumulate all of the symbols which are tested for and should return without calling the reader. Note that a version of such an "accumulator" procedure is required for each of the different versions of SID. To determine the exact form to be written it will probably be helpful to consult Appendices A and B. An "accumulator" procedure can be called by "syntax_error" to find out which symbols would be acceptable at the failed place in the syntax. A more complicated procedure to find sequences of acceptable symbols could be constructed in a similar manner but this does not seem to be useful for error recovery.

In the more general case where the error cannot be easily corrected by a single symbol insertion or deletion, the best approach is to skip past further input until some recognisable construct is reached and reset the position in the syntax accordingly. In many languages, statements follow semi-colons so a possible strategy is to skip to a semi-colon (and also skip the semi-colon itself) and set the syntax position to be the start of the rule for a statement sequence. This is done by marking such a position using a "mon" parameter for subsequent use. For example:

```
statseq = <recoverplace-mon> statseq stat,
          <recoverplace-mon> stat;
```

The mode MONITOR can be described as

```
MODE MONITOR = STRUCT(INT index, stind);
```

When the action "recoverplace" is called, the "index" field is the position in the "sid_code" of the next SID instruction, and the "stind" field is the current value of the analyser's internal stack pointer. An action may well be used to store a sequence of MONITOR values; recovery could then take place at one of these.

If a syntax error occurs, the (reference) fields "index" and "stind" of an INTERNALS value can be set to the corresponding fields of one of the stored MONITOR values; analysis will continue from the point in the syntax immediately after the action which stored the chosen MONITOR value. Note that the symbol in the "lex" field of the INTERNALS value must also be reset such that it equals the symbol to the right of the monitoring action. Care should be taken over the stacks as values will almost certainly have been pushed or popped between the recovery place and the failure.

The corresponding procedures might be something like:

```
MONITOR statement start;

PROC recover place = (MONITOR mon) BOOL:
statement start := mon;

PROC syntax error = (INTERNALS it) BOOL:
(... {output suitable SYNTAX ERROR message}
 ... {decide to skip to semicolon and continue with next statement}
 WHILE type OF lex OF it /= semicolon value
 DO lex OF it := next lex OD;
 {now read one more symbol so that lex is (hopefully) the
  first basic symbol of a statement}
 lex OF it := next lex;
 index OF it := index OF statement start;
      {set current position in syntax}
 stind OF it := stind OF statement start;
      {reset SID's internal stack to the correct level}
 TRUE {carry on with syntax analysis}
);
```

In practice recovery is unlikely to be quite so simple. Typically it might be necessary to have a stack of monitor points which are maintained as the various constructs in the syntax are worked through, and which for recovery are tried in an appropriate order, but the above example shows the general style.

5 Example of a Particular Syntax

This example describes all the steps necessary to run SID for a particular syntax. The syntax chosen is for a calculating procedure which takes a VECTOR [] CHAR as its parameter and delivers the result of the calculation. This example is simple and yet adequately indicates the technique.

The VECTOR [] CHAR parameter to the calculator provides the expression to be evaluated. The syntax of such a calculation may be described informally as follows:

$$sum_1;\ sum_2;\ sum_3;\ \ldots\ sum_n.$$

where the result of each sum is discarded at a semi-colon. A sum takes the form:

$$value_1? \bullet value_2? \bullet value_3? \ldots\ldots \bullet value_n?$$

where the question marks cause the value to be displayed and the •'s which may be "+", "-", "*" or "/" are evaluated from left to right. A value may itself be the result of a sub-calculation and takes the form:

$$number_1 \bullet number_2 \bullet number_3 \ldots\ldots \bullet number_n$$

where the •'s are evaluated according to the usual priority rules of arithmetic (i.e. "*" and "/" take precedence over "+" and "-"). It is also possible to include brackets in the normal way.

For example, the calculation:

"1+2?; (-1+2)*3? + 4*5? - -5+(7/8+9)? * (1+-2)?."

would first display "3" and then discard that value. The next display would again be "3" but this would then be added to the "20", which would then have "4" subtracted from it and the result multiplied by "-1" to give "-19" as a final answer.

A more formal description of the syntax of a calculation is required by SID and is:

```
{
BOOLSIZE 1          # maximum value of any basic symbol is less than the
                      number of bits in a single word #

BASICS

evaluate  (0)       # ? #
separate  (1)       # ; #
```

```
terminate (2)     # . #
number    (3)
orb       (4)     # ( #
crb       (5)     # ) #
plus      (6)     # + #
minus     (7)     # - #
multiply  (8)     # * #
divide    (9)     # / #

RULES

calculation = sum <return-pint> terminate,
              sum <separate-pint> separate calculation;

sum         = sum anyop value <opaction-pint-pint-pint--int>,
              value;

value       = value <evaluate-pint--int> evaluate,
              expression <evaluate-pint--int> evaluate;

expression  = expression addop term <opaction-pint-pint-pint--int>,
              term;

term        = term multop primary <opaction-pint-pint-pint--int>,
              primary;

primary     = <number-lv--int>number,
              addop primary <monadic-pint- int--int>,
              orb expression crb;

anyop       = addop,
              multop;

addop       = <operator-1--int>plus,
              <operator-2--int>minus;

multop      = <operator-3--int>multiply,
              <operator-4--int>divide;

ENTRY calculation
}
```

In the above syntax the actual representations of the basic symbols have been shown as a comment. Note that they are named in the SID input. There is only one stack (of INTs) which will hold intermediate results of arithmetic and also details of operators.

Examine the rule for "primary"; the first alternative caters for a literal number and in this case an action, also called "number", will be called with a LEXVAL parameter which will deliver the literal number (as an INT) and SID will cause it to be pushed onto the INT stack. Note that the action is written before the basic symbol "number" as the current lexical value should (if the input is syntactically correct) correspond to the symbol to the right of the current action.

The second alternative for primary is the case of a monadic operator (+ or -) preceding a primary. In this case the action "operator" in the rule for addop will have pushed an INT representing the operator (+ or -) and the rule for primary will also have pushed an INT. The action "monadic" removes two integers (the first will be the last pushed - i.e the primary, the second will be an integer representing the operator)

and delivers an INT which will be pushed on to the stack; this INT is the number obtained by applying the monadic operator to the previously evaluated primary.

The third alternative for primary is a bracketed expression. In this case the integer value of the expression will already be on the stack and so no further actions are required. The workings of the syntax and actions should become clear by examining the syntax together with the action procedures which follow.

At this stage the syntax could be checked by SID but there is not yet sufficient information to produce a compiled result. To achieve this a module must first be written to declare the modes and action procedures required. The following is a suitable text for such a module:

```
actions:
oneline :Module
intchars :Module
roll_m :Module

MODE LEXVAL = UNION(INT, VOID),
     LEX    = STRUCT(INT type, LEXVAL val),
     RESULT = INT;

PROC evaluate = (INT current value) INT:
BEGIN
  roll(oneline(("The expression currently evaluates to ",
                int chars(current value)
               ))
      );
  current value
END;

PROC monadic = (INT right, op) INT:
  CASE op IN right, - right ESAC;

PROC number = (LEXVAL lv) INT:
  CASE lv IN (INT value of number): value of number OUT 0 ESAC;

PROC opaction = (INT right, op no, left) INT:
  CASE op no
  IN left + right, left - right, left * right, left % right
  ESAC;

PROC operator = (INT op no) INT: op no;

PROC return = (INT result) RESULT: result;

PROC separate = (INT current value) VOID:  SKIP
CO simply throw away current value of calculation to start next one
   afresh CO


KEEP LEX, LEXVAL, RESULT, evaluate, monadic, number, opaction, operator,
     return, separate

FINISH
```

The above is compiled to form a module which is then added to the top of the syntax to form an Edfile to which "runsid" can be applied. This has been done below:

```
actions :Module

{
BOOLSIZE 1          # maximum value of any basic symbol is less than the
                      number of bits in a single word #

BASICS

evaluate  (0)       # ? #
separate  (1)       # ; #
terminate (2)       # . #
number    (3)
orb       (4)       # ( #
crb       (5)       # ) #
plus      (6)       # + #
minus     (7)       # - #
multiply  (8)       # * #
divide    (9)       # / #

RULES

calculation = sum <return-pint> terminate,
              sum <separate-pint> separate calculation;

sum         = sum anyop value <opaction-pint-pint-pint--int>,
              value;

value       = value <evaluate-pint--int> evaluate,
              expression <evaluate-pint--int> evaluate;

expression  = expression addop term <opaction-pint-pint-pint--int>,
              term;

term        = term multop primary <opaction-pint-pint-pint--int>,
              primary;

primary     = <number-lv--int>number,
              addop primary <monadic-pint-pint--int>,
              orb expression crb;

anyop       = addop,
              multop;

addop       = <operator-1--int>plus,
              <operator-2--int>minus;

multop      = <operator-3--int>multiply,
              <operator-4--int>divide;

ENTRY calculation
}
```

If the general Algol68 RS version is used and the result not automatically compiled the following is the contents of the Edfile produced:

```
analyser_m:
fail :Module

actions :Module

{
BOOLSIZE 1          # maximum value of any basic symbol is less than the
                      number of bits in a single word #

BASICS

evaluate  (0)       # ? #
separate  (1)       # ; #
terminate (2)       # . #
number    (3)
orb       (4)       # ( #
crb       (5)       # ) #
plus      (6)       # + #
minus     (7)       # - #
multiply  (8)       # * #
divide    (9)       # / #

RULES

calculation = sum <return-pint> terminate,
              sum <separate-pint> separate calculation;

sum         = sum anyop value <opaction-pint-pint-pint--int>,
              value;

value       = value <evaluate-pint--int> evaluate,
              expression <evaluate-pint--int> evaluate;

expression  = expression addop term <opaction-pint-pint-pint--int>,
              term;

term        = term multop primary <opaction-pint-pint-pint--int>,
              primary;

primary     = <number-lv--int>number,
              addop primary <monadic-pint-pint--int>,
              orb expression crb;

anyop       = addop,
              multop;

addop       = <operator-1--int>plus,
              <operator-2--int>minus;

multop      = <operator-3--int>multiply,
              <operator-4--int>divide;

ENTRY calculation
}

MODE INTERNALS = STRUCT(INT test_index,
                        VECTOR [0] CHAR sid_code,
                        REF INT index,stind,
                        REF LEX lex,
                        REF REF VECTOR [] INT sidstack,
                        VECTOR [0] BOOL blwds,
                        INT sid_mult);


PROC sid_convert = (VECTOR[]CHAR cblwds) VECTOR[]BOOL:
BEGIN
VECTOR [UPB cblwds * 8] BOOL blwds;
```

```
FOR char TO UPB cblwds
DO
   INT charasno := ABS cblwds[char];
   FOR bitno TO 8
   DO
      blwds[(char - 1) * 8 + bitno] := ODD charasno;
      charasno := charasno % 2
   OD
OD;

blwds

END;

VECTOR [] CHAR sid_code =
16r" 04 07 00 82 00 01 0f 00 01 b7 00 03 d4 00 07 00 82 00 01 20 00"
16r" 08 01 06 01 82 00 04 03 a9 00 07 00 82 00 01 2a 00 03 9a 00 07 00"
16r" 82 00 01 34 00 03 73 00 07 01 05 0b 01 5b 00 01 34 00 08 02 02"
16r" 06 04 05 0a 08 03 06 04 82 00 04 02 06 05 9b 00 04 01 20 00 06 06"
16r" 82 00 04 02 06 08 05 0a 08 04 06 08 82 00 04 02 06 07 8e 00 08 05"
16r" 06 07 82 00 04 02 07 02 02 00 01 82 00 01 34 00 08 06 03 73 00"
16r" 06 0a 05 0a 08 07 06 0a 82 00 04 02 06 09 8e 00 08 08 06 09 82 00"
16r" 04 02 07 01 02 00 01 5b 00 01 2a 00 08 06 03 9a 00 06 01 02 00"
16r" 08 01 06 01 82 00 04 03 a9 00 07 03 02 00 01 c6 00 01 0f 00 08 06"
16r" 03 b7 00 07 02 05 05 03 82 00 07 01 89 00 03 5b 00 06 02 05 0c"
16r" 08 09 06 02 82 00 04 03 02 00 06 03 90 00 09 01 06 03 82 00 04"
16r" 02";

 VECTOR [] CHAR sid_cblwds =
16r" d8 00 00"
16r" c0 00 00"
16r" 00 03 00"
16r" c0 03 00";

INT sid_mult = 24;

MODE INTSTACK = STRUCT (INTval, REF INTSTACK next);
REF INTSTACK intstack;


PROC sid_initstacks = VOID:
BEGIN
      intstack := NIL;
      SKIP
END;

COMMENT stacks created
1     INT
COMMENT

PROC sid_crash = (VECTOR [] CHAR ucstack) VOID:
BEGIN VECTOR [] CHAR mess1 = "Attempt to pop/top ",
                     mess2 = " value off empty stack";
      VECTOR [UPB mess1 + UPB mess2 + UPB ucstack] CHAR message;
      INT pos := UPB mess1;
      message[:pos] := mess1;
      message[pos + 1 : pos +:= UPB ucstack] := ucstack;
      message[pos + 1:] := mess2;
      fail(message)
END;

PROC sid_actions = (INT sid_no,
                    INT sid_index,
                    LEXVAL sid_lv,
                    INT sid_stind)VOID:
CASE sid_no
IN
   intstack := HEAP INTSTACK:=(evaluate(IF REF INTSTACK(intstack)
      IS NIL THEN sid_crash("INT"); SKIP ELSE INT d=val OF intstack;
      intstack := next OF intstack ; d FI),intstack),
   intstack := HEAP INTSTACK:=(monadic(IF REF INTSTACK(intstack)
      IS NIL THEN sid_crash("INT"); SKIP ELSE INT d=val OF intstack;
      intstack := next OF intstack ; d FI,
```

```
      IF REF INTSTACK(intstack) IS NIL THEN sid_crash("INT");
       SKIP ELSE INT d=val OF intstack;intstack := next OF intstack ; d FI),
      intstack),
   intstack := HEAP INTSTACK:=(number(sid_lv),intstack),
   intstack := HEAP INTSTACK:=(operator(2),intstack),
   intstack := HEAP INTSTACK:=(operator(1),intstack),
   intstack := HEAP INTSTACK:=(opaction(IF REF INTSTACK(intstack)
       IS NIL THEN sid_crash("INT"); SKIP ELSE INT d=val OF intstack;
      intstack := next OF intstack ; d FI,
      IF REF INTSTACK(intstack) IS NIL THEN sid_crash("INT");
       SKIP ELSE INT d=val OF intstack;intstack := next OF intstack ; d FI,
      IF REF INTSTACK(intstack) IS NIL THEN sid_crash("INT");
       SKIP ELSE INT d=val OF intstack;intstack := next OF intstack ; d FI),
      intstack),
   intstack := HEAP INTSTACK:=(operator(4),intstack),
   intstack := HEAP INTSTACK:=(operator(3),intstack),
   separate(IF REF INTSTACK(intstack) IS NIL THEN sid_crash("INT");
      SKIP ELSE INT d=val OF intstack;intstack := next OF intstack ; d FI),
   fail("Non-existent action called")
ESAC;

PROC sid_returns = (INT sid_no,
                    INT sid_index,
                    LEXVAL sid_lv,
                    INT sid_stind)RESULT:
CASE sid_no
IN
   return(IF REF INTSTACK(intstack) IS NIL THEN sid_crash("INT");
      SKIP ELSE INT d=val OF intstack;intstack := next OF intstack ; d FI),
   SKIP
ESAC;

PROC analyser = (PROC LEX reader,
                 PROC (INTERNALS) BOOL syntax_error
                ) UNION(VOID,RESULT):
( REF VECTOR [] INT sidstack := HEAP VECTOR [10] INT;
  INT stind := 0,index := 1;
  LEX lex;
  UNION(VOID,RESULT) result := EMPTY;
  VECTOR [] CHAR local sid_code = sid_code[:]; {the last 3 decs to diagnose}
  VECTOR [] BOOL blwds = sid_convert(sid_cblwds);

  sid_initstacks;

  DO
     CASE ABS sid_code[index]
     IN
        {call}
               (IF stind = UPB sidstack THEN
                   REF VECTOR [] INT x = HEAP VECTOR [UPB sidstack + 10] INT;
                   x[:UPB sidstack] := sidstack;
                   sidstack := x
                FI;
                sidstack[stind +:= 1] := index + 3;
                index := ABS sid_code[index+1] + ABS sid_code[index+2]*256
               ),

        {exit}
               (index := sidstack[stind];
                stind -:= 1
               ),

        {goto long}
               index:= ABS sid_code[index+1] + ABS sid_code[index+2]*256,

        {reader}
                (lex := reader;
                 index +:= 1
                ),

        {forward jump}
                index +:= ABS sid_code[index + 1],
```

```
        {skip if symbol = terminal}
                IF type OF lex + 1 = ABS sid_code[index + 1]
                THEN index +:= 4
                ELSE index +:= 2
                FI,

        {skip if symbol < terminal set}
                IF blwds[ABS sid_code[index + 1]*sid_mult + type OF lex + 1]
                THEN index +:= 4
                ELSE index +:= 2
                FI,

        {action}
                sid_actions(ABS sid_code[index+1],index+:=2,valOFlex,stind),

        {return}
                (result := sid_returns(ABS sid_code[index + 1],
                                       index+:=2, val OF lex, stind
                                      );
                 GOTO out
                )

    OUT
        {fail}
                IF NOT syntax_error((index - ABS sid_code[index] + 128
                                             - ABS sid_code[index + 1] * 128,
                                     sid_code,index,stind,lex,sidstack,blwds,
                                     sid_mult
                                    ))
                THEN GOTO out
                FI
    ESAC
  OD;

  out: result
)

KEEP analyser,INTERNALS
FINISH
```

The above text can be compiled to produce a module ("analyser_m") and together with the declarative module these two form the basis for the calculator procedure itself. This calculator procedure declares a lexical reader which is appropriate for calculator type applications but is generally not sufficient for more complex examples. Writing a lexical reader can be a time-consuming activity and so it makes sense to use or modify one which already exists. A number of simple lexical readers are described on Flex ("lex" is one); alternatively other Flex users have readers which may match other applications more closely. The advice here is to ask around before embarking solo on writing a new reader.

The text of the calculator procedure is:

```
calculate:
actions :Module
analyser_m :Module
oneline :Module
intchars :Module
fail :Module
```

```
PROC calculate = (VECTOR[]CHAR line)INT:
(INT index := 0;

 PROC lexical analyser = LEX:
 (CHAR ch; INT n;
  WHILE (index +:=1) <= UPB line ANDTH (ch := line[index]) = " "
  DO SKIP OD;
  IF index > UPB line
  THEN (2, EMPTY)
  ELIF ch >= "0" ANDTH ch <= "9"
  THEN n := ABS ch - ABS "0";
       WHILE (index+:=1) <= UPB line ANDTH
                (ch := line[index]) >= "0" ANDTH ch <= "9"
       DO n := n * 10 + ABS ch - ABS "0" OD;
       index -:= 1;
       (3, n)
  ELIF ch = "?" THEN (0, EMPTY)
  ELIF ch = ";" THEN (1, EMPTY)
  ELIF ch = "." THEN (2, EMPTY)
  ELIF ch = "(" THEN (4, EMPTY)
  ELIF ch = ")" THEN (5, EMPTY)
  ELIF ch = "+" THEN (6, EMPTY)
  ELIF ch = "-" THEN (7, EMPTY)
  ELIF ch = "*" THEN (8, EMPTY)
  ELIF ch = "/" THEN (9, EMPTY)
  ELSE fail(oneline(("Unknown symbol at character position ",
                     intchars(index)))); SKIP
  FI);

 PROC syntax error = (INTERNALS error pos)BOOL:
 (fail(oneline(("Syntax error at or before character position ",
                intchars(index))));
  FALSE);

 CASE analyser(lexical analyser, syntax error)
 IN (INT result): result
 OUT SKIP
 ESAC)

KEEP calculate
FINISH
```

This is compiled and used to form the (final) module and by applying "file68" to this module, a Flex procedure of mode "Filed(Vec Char -> Int)" can be constructed.

Acknowledgements

The authors would particularly like to thank J.M. Foster who wrote the transformational content of SID, and I.F. Currie who was responsible for the form of the SID output. Acknowledgement is also due to many others at RSRE, too numerous to name, who have at some time contributed toward the evolution of SID.

References

[1] Foster J.M., "A Syntax Improving Program", Computer Journal, vol 2, pp31, 1968.

[2] Woodward P.M. & Bond S.G., "Guide to ALGOL 68 (for users of RS systems)", Edward Arnold, 1983.

[3] Currie I.F., Foster J.M. & Edwards P.W., "Perq Flex Firmware", RSRE Report 85015, 1985.

[4] Foster J.M., Currie I.F. & Edwards P.W., "Flex: A Working Computer with an Architecture Based on Procedure Values", RSRE Memorandum 3500, 1982.

[5] Currie I.F., Edwards P.W. & Foster J.M., "Kernel and System Procedures in Flex", RSRE Memorandum 3626, 1983.

Appendices

A The Form of the SID Output for Version A

This version uses several features of the Flex architecture not generally available to Algol68 programs on ordinary machines. Hence this version can only be used on Flex.

That part of the output that varies according to the input syntax consists of two vectors of characters. The first, called "sid_code" contains a sequence of instructions; the second called "sid_cblwds" contains the boolean words used for indicating multiple legal basic symbols.

The "sid_code" vector of characters contains a sequence of "instructions" of the form given below. Note that it is not very densely packed; each number below is represented as a single character.

1 u v Call rule at index u + 256*v in sid_code (stacking current position).

2 Exit from current rule returning to calling rule.

3 u v Goto index u + 256 * v in sid_code.

4 Call reader to get next basic symbol.

5 u Relative jump by u.

6 u Skip next two characters if value of current basic symbol is equal to u - 1.

7 u Skip next two characters if current basic symbol is contained in boolean word number u. (The Boolean words are numbered from 0).

8 u {followed by params - see below}
  Call action number u.

m n {m > 128}

  Fail. Start of test or cascade of tests which failed was at index m - 128 + n * 128 before current position.

The form of parameters to action calls is a sequence of the following:

4 u Parameter is literal u
5 0 Parameter is LEXVAL

6 0 Parameter is MON
7 u Parameter obtained by popping stack u
8 u Parameter obtained from top of stack u

terminated with one of the following:

1 Last action (i.e. return))
2 Action with no result) End of parameters
3 u Push result on stack u)

At points in the syntax there may be several legal alternative lexical values; for each such point, there is boolean word which indicates which symbols those are. A boolean word is a group of bits - there is one bit for every basic symbol- such that bit_n is set if the basic symbol with characteristic value *n* is legal at the point in question (bits are numbered from 0).

Each boolean word is "packed" into a few characters such that the least significant bit of the first character represents basic symbol zero. As a radix string is a convenient denotation to use in an automatically generated output, the sets of characters formed from each boolean word are further "packed" into the vector "sid_cblwds". However, to facilitate access to each boolean word from within "analyser", "sid_cblwds" is converted into a vector of booleans, "sid_blwds".

The part of the output that does not depend on the input syntax consists of two Edfiles which between them define the analyser. The first Edfile contains the following:

```
MODE STACK = STRUCT(REF VECTOR [] INT head, REF STACK tail),
             INTERNALS = STRUCT(INT test_index,
                                VECTOR [0] CHAR sid_code,
                                REF INT index,stind,
                                REF LEX lex,
                                REF REF VECTOR [] INT sidstack,
                                VECTOR [0] BOOL blwds,
                                INT sid_mult
                               );


PROC sid_convert = (VECTOR[]CHAR cblwds) VECTOR[]BOOL:
BEGIN
VECTOR [UPB cblwds * 8] BOOL blwds;

FOR char TO UPB cblwds
DO
   INT charasno := ABS cblwds[char];
   FOR bitno TO 8
   DO
      blwds[(char - 1) * 8 + bitno] := ODD charasno;
      charasno := charasno % 2
   OD
OD;

blwds
```

```
END;
```

---

and the second contains:

```
OP (INT) REF VECTOR [] INT PACK = BIOP 1266;
OP (LEXVAL) REF VECTOR [] INT PACK = BIOP 1266;
OP (VECTOR [] INT) INT UNPACK = BIOP 1267;

PROC analyser = (PROC LEX reader,
                 PROC (INTERNALS) BOOL syntax_error
                ) UNION(VOID,RESULT):
( INT i,j,k;
  PROC (INT) INT action;
  VECTOR [] BOOL blwds = sid_convert(cbwds);
  REF VECTOR [] INT sidstack := HEAP VECTOR [0] INT;
  REF VECTOR [] INT x;
  INT stind := 0, index := 1;
  BOOL void_result := FALSE;
  LEX lex;
  VECTOR [] CHAR local sid_code = sid_code[:];       {to enable diagnosing}
  FORALL st IN stacks DO st := NIL OD;
  DO CASE ABS sid_code[index]
     IN
    { call }
      (IF stind = UPB sidstack
       THEN REF VECTOR [] INT x = HEAP VECTOR [UPB sidstack+10] INT;
            x[:UPB sidstack] := sidstack;
            sidstack := x
       FI;
       sidstack[stind +:= 1] := index + 3;
       index := ABS sid_code[index+1] + ABS sid_code[index+2]*256
      ),

    { exit }
      (index := sidstack[stind]; stind -:= 1),

    { goto long }
       index := ABS sid_code[index+1] + ABS sid_code[index+2]*256,

    { reader }
      (lex := reader; index +:= 1),

    { forward jump }
       index +:= ABS sid_code[index+1],

    { skip if symbol = terminal }
       IF type OF lex + 1 = ABS sid_code[index+1]
       THEN index +:= 4
       ELSE index +:= 2
       FI,

    { skip if symbol < terminal set }
       IF blwds[type OF lex + 1 + ABS sid_code[index+1] * sidmult]
       THEN index +:= 4
       ELSE index +:= 2
       FI,

    { action }
     (i := index + 2;
      j := 0;
      WHILE CASE ABS sid_code[i]
            IN (k:=-1; FALSE),
               (k := 0; FALSE),
               (k := ABS sid_code[i +:= 1]; FALSE),
               (j +:= 1; TRUE),
               (j +:= lexval_size; TRUE),
                              {must be incremented by size of LEXVAL}
```

```
            (j +:= 2; TRUE)
        OUT INT st = ABS sid_code[i+1];
            REF STACK s = stacks[st];
            IF s IS REF STACK(NIL)
            THEN fail(oneline(("Stack ", intchars(st), " empty")))
            FI;
            j +:= UPB head OF s; TRUE
        ESAC
    OD i +:= 2 OD;
    x := HEAP VECTOR [j] INT;
    j := 0;
    FOR w FROM index + 3 BY 2 TO i - 1
    DO INT u1 = ABS sid_code[w - 1], u2 = ABS sid_code[w];
       IF u1 <= 5
       THEN IF u1 = 4
            THEN x[j +:= 1] := u2
            ELSE {vector size must = LEXVAL}
                 x[j + 1 : j +:= lexval_size] := PACK val OF lex
            FI
       ELIF u1 = 6
       THEN x[j +:= 1] := i + 1; x[j +:= 1] := stind
       ELSE REF STACK s = stacks[u2];
            IF s IS REF STACK(NIL)
            THEN fail(oneline(("Stack ", intchars(u2), " empty")))
            FI;
            x[j + 1 : j +:= UPB head OF s] := head OF s;
            IF u1 = 7 THEN stacks[u2] := tail OF s FI
       FI
    OD;
    action := actions[ABS sid_code[index+1]];
    IF k = 0
    THEN action(UNPACK x)
    ELIF k < 0
    THEN GOTO out
    ELSE REF VECTOR [] INT ans = PACK action(UNPACK x);
         stacks[k] := HEAP STACK := (ans,stacks[k])
    FI;
    index := i + 1
  )

  OUT
 { fail }
    IF NOT
    syntax_error((index-ABS sid_code[index]+128 -
                                          ABS sid_code[index+1]*128,
                  sid_code,index,stind,lex,sidstack,blwds,sidmult
                 )
                )
    THEN void_result := TRUE; GOTO out
    FI
 ESAC
OD;

out: IF void_result THEN EMPTY ELSE (Y action)(UNPACK x) FI
)

KEEP analyser,INTERNALS

FINISH
```

B The Form of the SID Output for Version B

This is a general version of SID which produces an analyser which should run on any Algol68 RS system.

That part of the output that varies according to the input syntax consists of two vectors of characters. The first, called "sid_code" contains a sequence of instructions; the second called "sid_cblwds" contains the boolean words used for indicating multiple legal basic symbols.

The "sid_code" vector of characters contains a sequence of "instructions" of the form given below. Note that it is not very densely packed. Each *number* below is represented as a single character.

1 u v Call rule at index u + 256*v in sid_code (stacking current position).

2 Exit from current rule returning to calling rule.

3 u v Goto index u + 256 * v in sid_code.

4 Call reader to get next basic symbol.

5 u Relative jump by u

6 u Skip next two characters if value of current basic symbol is equal to u - 1.

7 u Skip next two characters if current basic symbol is contained in boolean word number u. (The Boolean words are numbered from 0)

8 u Call action number u (not the return action)
{Note that different forms of parameters for the same procedure will have different numbers in this version of SID}

9 u Call the return action version u.
{Note that different forms of parameters for the return action will have different numbers in this version of SID}

m n {m > 128}

Fail. Start of test or cascade of tests which failed was at index m - 128 + n * 128 before current position.

The form of parameters to action calls is not included in the sid_code as this information in included directly in the analyser produced.

At points in the syntax there may be several legal alternative lexical values; for each such point, there is boolean word which indicates which symbols those are. A boolean word is a group of bits - there is one bit for every basic symbol- such that bit_n is set if the basic symbol with characteristic value *n* is legal at the point in question (bits are numbered from 0).

Each boolean word is "packed" into a few characters such that the least significant bit of the first character represents basic symbol zero. As a radix string is a convenient denotation to use in an automatically generated output, the sets of characters formed from each boolean word are further "packed" into the vector "sid_cblwds". However, to facilitate access to each boolean word from within "analyser", "sid_cblwds" is converted into a vector of booleans, "sid_blwds".

The part of the output that does not depend on the input syntax consists of two Edfiles which between them define the analyser. The first Edfile contains the following:

```
MODE INTERNALS = STRUCT(INT test_index,
                        VECTOR [0] CHAR sid_code,
                        REF INT index,stind,
                        REF LEX lex,
                        REF REF VECTOR [] INT sidstack,
                        VECTOR [0] BOOL blwds,
                        INT sid_mult);


PROC sid_convert = (VECTOR[]CHAR cblwds) VECTOR[]BOOL:
BEGIN
VECTOR [UPB cblwds * 8] BOOL blwds;

FOR char TO UPB cblwds
DO
   INT charasno := ABS cblwds[char];
   FOR bitno TO 8
   DO
      blwds[(char - 1) * 8 + bitno] := ODD charasno;
      charasno := charasno % 2
   OD
OD;

blwds

END;
```

and the second contains:

```
PROC analyser = (PROC LEX reader,
                 PROC (INTERNALS) BOOL syntax_error
```

```
                ) UNION(VOID,RESULT):
( REF VECTOR [] INT sidstack := HEAP VECTOR [10] INT;
  INT stind := 0,index := 1;
  LEX lex;
  UNION(VOID,RESULT) result := EMPTY;
  VECTOR [] CHAR local sid_code = sid_code[:]; {the last 3 decs to diagnose}
  VECTOR [] BOOL blwds = sid_convert(sid_cblwds);

  sid_initstacks;

  DO
     CASE ABS sid_code[index]
     IN
        {call}
               (IF stind = UPB sidstack THEN
                   REF VECTOR [] INT x = HEAP VECTOR [UPB sidstack + 10] INT;
                   x[:UPB sidstack] := sidstack;
                   sidstack := x
                FI;
                sidstack[stind +:= 1] := index + 3;
                index := ABS sid_code[index + 1] +
                                                ABS sid_code[index + 2] * 256
               ),

        {exit}
               (index := sidstack[stind];
                stind -:= 1
               ),

        {goto long}
               index:= ABS sid_code[index+1] + ABS sid_code[index+2] * 256,

        {reader}
                (lex := reader;
                 index +:= 1
                ),

        {forward jump}
                index +:= ABS sid_code[index + 1],

        {skip if symbol = terminal}
                IF type OF lex + 1 = ABS sid_code[index + 1]
                THEN index +:= 4
                ELSE index +:= 2
                FI,

        {skip if symbol < terminal set}
                IF blwds[ABS sid_code[index + 1] * sid_mult + typeOFlex + 1]
                THEN index +:= 4
                ELSE index +:= 2
                FI,

        {action}
                sid_actions(ABS sid_code[index+1],index+:=2,valOFlex,stind),

        {return}
                (result := sid_returns(ABS sid_code[index + 1],
                                       index+:=2, val OF lex, stind
                                      );
                 GOTO out
                )

     OUT
        {fail}
                IF NOT syntax_error((index - ABS sid_code[index] + 128
                                           - ABS sid_code[index + 1] * 128,
                                     sid_code,index,stind,lex,sidstack,blwds,
                                     sid_mult
                                    ))
                THEN GOTO out
                FI
     ESAC
  OD;
```

```
  out: result
)

KEEP analyser,INTERNALS
FINISH
```

C The Syntax of the Input to SID

The syntax for SID's input is given below together with some annotations. When using BOOLSIZE *note* that the Logica Flex has a 24-bit word and the ICL Perq2 Flex *also* has, in effect, a 24-bit word.

```
{

BOOLSIZE 1              # This specifies how many words are required in
                          order to have enough bits for the basic symbol
                          with the largest characteristic value. This
                          information may be omitted in which case a
                          BOOLSIZE of 3 is assumed by default #

BASICS                  # list of basic symbols #

basics (2)              # BASICS #
rules  (3)              # RULES  #
entry  (6)              # ENTRY  #
ident  (7)
ord    (0)              # ( - note basic symbols can be numbered from zero #
crd    (1)              # ) #
comma  (4)              # , #
int    (8)
minus  (5)              # - #
semi   (19)             # ; #
eq     (21)             # = #
ls     (20)             # < #
gt     (22)             # > #
dollar (23)             # in a 24 bit word this is the largest basic symbol
                          allowed with BOOLSIZE 1 #
endsyntax (9)

RULES

input = basics basiclist rules rulelist entry ident endsyntax;


basiclist = basic,
            basic basiclist;

basic     = ident,
            ident ord defs crd;

defs      = def,
            def comma defs;  # a basic symbol can have several
                               numbers #

def       = int,
            ident,
            minus def;       # all but the particular number #

rulelist  = rule,
            rule semi rulelist;

rule      = ,
            ident eq alts;

alts      = alt0,
            alt0 comma alts;

alt0      = dollar,
            alt;

alt       = ,                # empty alternative #
            item alt;

item      = fn,
            ident;
```

```
fn        = ls ident gt;      # the ident describes an action procedure
                                as specified in section 2 #

ENTRY input           # the start rule for the syntax #

}
```

DOCUMENT CONTROL SHEET

Overall security classification of sheet ....UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Memorandum 3768	3. Agency Reference	4. Report Security Classification U/C
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location ROYAL SIGNALS AND RADAR ESTABLISHEMNT		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title

A GUIDE TO SID FOR USERS OF THE FLEX COMPUTER

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
GOODENOUGH S J	TAYLOR P D	WHITAKER G D		

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

UNLIMITED

Descriptors (or keywords)

continue on separate piece of paper

Abstract This document describes a utility know as "SID" that is available on the RSRE Flex Computer. SID accepts a syntax with embedded actions, transforms it, and produces an Algo 168 RS module (suitable for any RS system) that contains an analyser procedure. This analyser will check whether an input conforms to the given syntax and, while doing so, will initiate calls of the embedded actions which take the form of user-defined procedures that operate on stacks managed by the analyser. By combining the analyser module with other user-defined modules to form a complete program, syntax-directed utilities such as compilers, interpreters and translators may be constructed; indeed SID was written using itself.

END

DTIC

7-86