Static semantic features of ALGOL60 and BASIC

M. H. Williams

Department of Computer Science, Rhodes University, Grahamstown 6140, South Africa

The static semantic rules of ALGOL60 and BASIC are expressed in a formal notation to demonstrate that the notation can be used for such languages. This is seen as an aid to both the compiler writer and the user, partly in that the intentions of the language designer can be communicated clearly and unequivocally and partly in that the specification may be used as a guide to producing a correct compiler or in verifying the correctness of a compiler.

(Received October 1975)

1. Introduction

In a previous paper (Williams, 1978) a formal notation was presented for describing the static semantic rules, or 'context-sensitive' rules (Ledgard, 1969) of a language. This notation was developed to interface with the BNF specification of the syntax of a language. The notation is based on the use of strings (either those represented by metalinguistic variables in the BNF specification or literal strings) and stacks. It is intended that such a notation should provide the compiler writer with a guide for the implementation of the language or give the user an understanding of the details of the language.

In this paper the notation is used to describe the static semantic rules of two programming languages: ALGOL60 and BASIC. In the case of ALGOL60, the static semantic rules are complex and have never been completely specified in a useful notation for the compiler writer. BASIC, on the other hand, has relatively simple static semantic rules although there is no general agreement on the syntax rules.

The fact that the static semantic rules for these two languages can be expressed completely in this notation means that the language designer can communicate his intentions directly and unambiguously to the compiler writer and the user, rather than attempting to convey his ideas by means of a set of English sentences which may often give rise to misinterpretation. This notation also gives the compiler writer a clear guide as to how these rules may be implemented.

2. BASIC

Unlike ALGOL60 there is no generally accepted syntax specification for BASIC. Bull *et al.* (1973) have produced a detailed specification which includes all the facilities of a full BASIC compiler. However, as many BASIC compilers do not provide all these facilities and since the specification is a little large to handle here, a specification of a 'basic' subset of the language was sought.

Lee (1972) provides such a specification. The subset of BASIC which he defines does not include strings, matrix statements, subprograms, file handling or advanced input/output facilities. After correction of a few minor errors in Lee's specification and conversion of his context-sensitive rules to a context-free form, one arrives at a specification which is not a strict subset of Bull's more general definition. Thus changes were made to bring the specification more into line with Bull's definition (e.g. null programs or empty DATA statements are not valid, whereas parameterless functions and generalised forms of IF and ON statements are permitted).

The syntax specifications given by both Lee and Bull rely on statements being in correct order of line number. However, this is not very helpful for the compiler writer who is implementing an interactive BASIC system. He needs a specification which will check that each statement has the correct form as it is read in (whatever the order in which statements are read). Thus the syntax rules were changed slightly to cater for this. Finally the resulting set of rules was converted into 'standard' BNF notation.

The static semantic rules for BASIC can be summarised as follows:

1. Line numbers may be entered in any order; duplicate line numbers and null lines may occur. The lines must be sorted into line number order, and in the case of duplicate lines, the last supplied is used.
2. The same letter may be used both as a simple variable and as a one or two dimensional array. An array is declared either explicitly in a DIM statement or implicitly by its first appearance in the program. In the case of explicit declaration all references to the array (subscripted variables) must have the same number of subscripts as there are dimensions in the declaration; if declaration is implicit all references to the array must have the same number of subscripts as the first reference to the array. A DIM statement can occur anywhere in a program.
3. The same array name cannot be used to apply to more than one array in a program.
4. If a function is called in a program, the actual parameters supplied must correspond in number, type (relevant when extending the definition of BASIC to include strings and string variables) and order to the formal parameters in the function definition. A function definition may occur anywhere in a program, not necessarily at a lower-numbered line than that of the function call.
5. A function name may be used as a destination for a LET statement only within the definition block for that function.
6. Function definitions may not be nested.
7. Control may not be transferred from outside a function definition to an intermediate line within it, nor may control be transferred out of a function definition. However, control may be passed to the first or only line of a definition block.
8. Each FOR statement must have a corresponding NEXT statement which refers to the same control variable. FOR-NEXT loops may not overlap.
9. A FOR-NEXT loop may not overlap with a function definition.
10. For each multiline definition statement DEF <user function>, there must be a corresponding FNEND statement.
11. If a user function is called in a program, the function must be defined somewhere in the program.
12. If a line number is used as a destination line number in a

program, it must also occur as a line no def somewhere in the program.

The latter two rules may be regarded as unnecessary. However, as some BASIC compilers perform these checks at compile time and some do not, they have been included in this specification on the grounds that they can easily be removed if not required.

The syntax interpretation rules are divided into two passes. The first pass handles rule 1 above. This accepts lines in any order, checks that the form of each line is correct and stores it (overwriting any previous line with the same number). On encountering the END statement, the program is complete and lines are sorted into ascending order for the second pass. The concatenation operation is used to assemble the program in the correct order for the second pass. In practice the level of reconstruction will obviously depend on the compiler writer.

The second pass uses the following stacks:

(*a*) AS (array stack) with entries of form:
(array name, number of dimensions, declared or undeclared).

(*b*) APL (actual parameter list) with entries of form:
type of actual parameter.

(*c*) PARS (parameter stack) with entries of form:
(number of parameters, stack of parameter types, declared or undeclared).

(*d*) FS (function stack) with entries of form:
(function name, number of parameters, stack of parameter types, declared or undeclared).

(*e*) FPL (formal parameter list) with entries of form:
type of formal parameter.

(*f*) UFS (user function stack) should either be empty or contain the name of the current user function being defined.

(*g*) ULOD (undefined line numbers outside definitions)

(*h*) LOD (line numbers outside definitions)

(*i*) ULWCD (undefined line numbers within current definition)

(*j*) LWCD (line numbers within current definition)

} have entries of form: line number.

(*k*) CURLINE (current line) contains the line number of the current line.

(*l*) LS (line number stack) contains an entry for each line number in the program.

(*m*) FORS (for stack) has an entry of form: (simple variable) for each unmatched FOR statement encountered thus far in the current function definition or outer block.

(*n*) DUMP contains the contents of the for stack while a function definition is being handled.

Each entry (parameter type) in APL and FPL will always be 'A' (arithmetic) for this specification so that mismatched types cannot occur. However, when extending the language to include strings, entries will be either 'A' or 'S' and hence matching parameter types becomes important.

The complete specification of syntax and static semantic rules follows.

2.1 *Syntax rules for BASIC*

1. < program > ::= < statement sequence > < terminal st >
2. < statement sequence > ::= < statement > | < statement > < statement sequence >
3. < statement > ::= < line no def > < unnumbered st > < new line > | < line no def > < new line >
4. < unnumbered st > ::= < simple statement > | < single line def st > | < multiline def st > | < fn end st >
5. < simple statement > ::= < let st > | < read st > | < data st > | < print st > | < goto st > | < on st > | < if st > | < for st > | < next st > | < dim st > | < gosub st > | < return st > | < restore st > | < stop st > | < rem st >
6. < terminal st > ::= < line no def > END < new line >
7. < line no def > ::= < line no >
8. < line no > ::= < digit > | < digit > < digit > | < digit > < digit > < digit > | < digit > < digit > < digit > < digit > | < digit > < digit > < digit > < digit > < digit >
9. < expression > ::= < plusminus > < term > | < term > | < expression > < plusminus > < term >
10. < term > ::= < factor > | < term > < timesover > < factor >
11. < factor > ::= < primary > | < factor > ↑ < primary >
12. < plusminus > ::= + | −
13. < timesover > ::= × | /
14. < primary > ::= < constant > | < variable > | < function ref > | (< expression >)
15. < variable > ::= < simple variable > | < subscripted variable >
16. < simple variable > ::= < letter > | < letter > < digit >
17. < subscripted variable > ::= < letter > (< expression >) | < letter > (< expression > , < expression >)
18. < integer > ::= < digit > | < integer > < digit >
19. < number > ::= < integer > | . < integer > | < integer > . < integer >
20. < exponent > ::= E < plusminus > < expt > | E < expt >
21. < expt > ::= < digit > | < digit > < digit >
22. < constant > ::= < number > < exponent > | < number >
23. < digit > ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
24. < function ref > ::= < function name > | < function name > (< actual parameter list >)
25. < function name > ::= < library function > | < user function >
26. < actual parameter list > ::= < expression > | < actual parameter list > , < expression >
27. < user function > ::= FN < letter >
28. < library function > ::= SIN | COS | TAN | ATN | EXP | ABS | LOG | SQR | INT | RND
29. < dim st > ::= DIM < array dimension list >
30. < array dimension list > ::= < array dim > | < array dim > , < array dimension list >
31. < array dim > ::= < letter > (< integer >) | < letter > (< integer > , < integer >)
32. < let st > ::= LET < destination > = < expression > | < destination > = < expression >
33. < destination > ::= < variable > | < user function >
34. < signed number > ::= < plusminus > < constant > | < constant >
35. < data list > ::= < signed number > | < data list > , < signed number >
36. < data st > ::= DATA < data list >
37. < restore st > ::= RESTORE
38. < goto st > ::= GOTO < destination line no >
39. < on st > ::= ON < expression > < designator > < line number list >
40. < designator > ::= THEN | GOTO | GOSUB
41. < line number list > ::= < destination line no > | < line number list > , < destination line no >
42. < gosub st > ::= GOSUB < destination line no >
43. < return st > ::= RETURN
44. < if st > ::= IF < condition > < designator > < destination line no >
45. < destination line no > ::= < line no >

46. < condition > ::= < expression > < relation >
< expression >
47. < relation > ::= = | > = | < = | > | < | < >
48. < for st > ::= FOR < simple variable > = < expression >
TO < expression > |FOR < simple variable > =
< expression > TO < expression > STEP < expression >
49. < next st > ::= NEXT < simple variable >
50. < rem st > ::= REM < comment > |REM
51. < comment > ::= < character > |
< comment > < character >
52. < stop st > ::= STOP
53. < read st > ::= READ < destination list >
54. < destination list > ::= < variable > |
< destination list >, < variable >
55. < print st > ::= PRINT < print list >
56. < item sep print list > ::=
< item sequence > < separator sequence > |
< item sep print list > < item sequence >
< separator sequence >
57. < sep item print list > ::=
< separator sequence > < item sequence > |
< sep item print list > < separator sequence >
< item sequence >
58. < item list > ::= < item sequence > | < item sep print list > |
< item sep print list > < item sequence >
59. < sep list > ::= < separator sequence > |
< sep item print list > |
< sep item print list > < separator sequence >
60. < print list > ::= < item list > | < sep list >
61. < separator sequence > ::= < print separator > |
< separator sequence > < print separator >
62. < print separator > ::= , |;
63. < item sequence > ::= < message > | < print expression > |
< message > < print expression > |
< item sequence > < message > |
< item sequence > < message > < print expression >
64. < print expression > ::= < expression > |
TAB (< expression >)
65. < message > ::= " < character string > "
66. < character string > ::= < character > |
< character string > < character >
67. < character > ::= < letter > | < digit > |
< special character >
68. < def head > ::= DEF < user function > (< formal par
list >) | DEF < user function >
69. < formal par list > ::= < simple variable > |
< formal par list >, < simple variable >
70. < single line def st > ::= < def head > = < expression >
71. < multiline def st > ::= < def head >
72. < fn end st > ::= FNEND
73. < letter > ::= A |B |C |D |E |F |G |H |I |J |K |L |M |N |O |P |
Q |R |S |T |U |V |W |X |Y |Z
74. < special character > ::= + | − | × |/ | = |) |(| > | < |. |; |
< blank > |$ |↑ | ? |,
75. < newline > ::= **CR**

2.2 *Static semantic rules for pass I*

3. < statement > ::= < line no def > < unnumbered st >
< newline > {𝒮(SS, < line no def >, < unnumbered st > →
Comp2,(< line no def >, < unnumbered st >)↓SS} |
< line no def > < newline > {𝒮(SS, < line no def > ,λ→
Comp2,(< line no def > ,λ)↓SS)}
6. < terminal st > ::= < line no def > END < newline >
{𝒮(SS, < line no def > ,'END'→Comp2,(< line no def >,
'END')↓SS)}
1. < program > ::= < statement sequence > < terminal st >
{𝒜(SS,SS);λ→x;∀SS:(y↑SS;x.Comp1(y).Comp2(y).
'CR'→x;Comp1(y)↓LS);x→ < program > }

2.3 *Static semantic rules for pass II*

17. < subscripted variable > ::= < letter > (< expression >)
{𝒮(AS, < letter >, **if** Comp2≠'1' **then** (**if** Comp3='U'
then E1 **else** E2),
(< letter > ,'1','U')↓AS)} |
< letter > (< expression >, < expression >)
{𝒮(AS, < letter >, **if** Comp2≠'2' **then** (**if** Comp3='U'
then E1 **else** E2),
(< letter > ,'2', 'U')↓AS)}
31. < array discussion > ::= < letter > (< integer >)
{𝒮(AS, < letter >, **if** Comp3='D' **then** E3
else if Comp2≠"1" **then** E4 **else** 'D'→Comp3,
(< letter >, '1', 'D')↓AS)} |
< letter > (< integer >, < integer >)
{𝒮(AS, < letter >, **if** Comp3='D' **then** E3 **else if**
Comp2≠'2' **then** E4 **else** 'D'→Comp3,
(< letter > ,'2','D')↓AS)}
26. < actual parameter list > ::= < expression > {'A'↓APL} |
< actual parameter list >, < expression > {**ditto**}
28. < library function > ::= SIN{('1',('A'),'D')↓PARS} |
COS{**ditto**} |TAN{**ditto**} |ATN{**ditto**} |EXP{**ditto**} |
ABS{**ditto**} |LOG{**ditto**} |SQR{**ditto**} |INT{**ditto**} |
RND{**ditto**}
25. < function name > ::= < library function > | < user
function > {𝒮(FS, < user function > ,(Comp2,Comp3,
Comp4)↓PARS,('U',0,0)↓PARS)}
24. < function ref > ::= < function name > {x↑PARS; **if**
Comp1(x)='U' **then** (< function name > ,0,0,'U')↓FS
else if Comp1(x)≠0 **then** (**if** Comp3='D' **then** E5 **else**
E6)} |
< function name > (< actual parameter list >){x↑PARS;
if Comp1(x)='U' **then** (< function name > ,𝒩(APL),
APL,'U')↓FS **else if** Comp1(x)≠𝒩(APL) **then** (**if**
Comp3='D' **then** E5 **else** E6)
else (Comp2(x)→y;∀y:(u↑y;v↑APL; **if** u≠v **then** E7));
λ ⇑ APL}
69. < formal par list > ::= < simple variable > {'A'↓FPL;
𝒮(T, < simple variable > ,E20, < simple variable > ↓T)} |
< formal par list >, < simple variable > {**ditto**}
68. < def head > ::=
DEF < user function > (< formal par list >)
{**if** 𝒩(UFS)≠0 **then** E8;𝒮(FS, < user function >, **if**
Comp4='D' **then** E9 **else** 'D'→Comp4;
if Comp2≠𝒩(FPL) **then** E10;Comp3→y;
∀y:(x↑y;z↑ FPL; **if** x≠z **then** E11),
(< user function > ,𝒩(FPL),FPL,'D')↓FS);
< user function > ↓UFS;λ ⇑ T} |
DEF < user function >
{**if** 𝒩(UFS)≠0 **then** E8;𝒮(FS, < user function >, **if**
Comp4='D' **then** E9 **else** 'D'→Comp4; **if** Comp2≠0
then E10,(< user function > ,0,0,'D')↓FS); < user
function > ↓UFS}
70. < single line def st > ::= < def head > = < expression >
{λ↑UFS}
71. < multiline def st > ::= < def head > {DUMP ⇑ FORS}
7. < line no def > ::= < line no >
{**if** 𝒩(UFS)=0 **then** (𝒮(ULOD, < line no > ,λ→Comp1,
−); < line no > ↓LOD) **else** (𝒮(ULWCD, < line no >,
λ→Comp1, −); < line no > ↓LWCD); **if** 𝒩(CURLINE)
≠0 **then** λ↑CURLINE; < line no > ↓CURLINE}
45. < destination line no > ::= < line no >
{𝒮(LS, < line no >, −, E12); **if** < line no > < = CURLINE
then (**if** 𝒩(UFS)=0 **then** 𝒮(LOD, < line no >, −, E13)
else 𝒮(LWCD, < line no >, −, E14)) **else** (**if** 𝒩(UFS)=0
then 𝒮(ULOD, < line no >, −, < line no > ↓ULOD)
else 𝒮(ULWCD, < line no >,
−, < line no > ↓ ULWCD)}
72. < fn end st > ::= FNEND{λ↑UFS;∀ULWCD:
(x↑ULWCD; **if** x≠λ **then** E14);λ ⇑ LWCD; **if** 𝒩(FORS)

$\neq 0$ **then** E17;DUMP $\Downarrow$ FORS;$\lambda\uparrow$DUMP}

33. < destination > ::= < variable > | < user function >
{**if** < user function > $\neq \mathcal{T}$(UFS) **then** E15}

48. < for st > ::= FOR < simple variable > = < expression >
TO < expression > { < simple variable > $\downarrow$FORS} |
FOR < simple variable > = < expression >
TO < expression > STEP < expression > {**ditto**}

49. < next st > ::= NEXT < simple variable >
{**if** $\mathcal{N}$(FORS) = 0 **then** E16 **else** (x$\uparrow$FORS; **if** x $\neq$ < simple variable > **then** E16)}

1. < program > ::= < statement sequence > < terminal st >
{$\forall$FS:(x$\uparrow$FS; **if** Comp4(x) = 'U' **then** E18); **if** $\mathcal{N}$(FORS) $\neq 0$ **then** E17; **if** $\mathcal{N}$(ULOD) $\neq 0$ **then** E13; **if** $\mathcal{N}$(UFS) $\neq 0$ **then** E19}

2.4 *Error numbers*

1. Number of subscripts of subscripted variable does not match number of subscripts previously used.
2. Number of subscripts of subscripted variable does not match number of subscripts declared in DIM statement.
3. Array defined twice.
4. Number of subscripts in array declaration does not match number of subscripts previously used.
5. Number of actual parameters in a function call does not match number of formal parameters in function definition.
6. Number of actual parameters in a function call differs from number of actual parameters used in a previous call to the same function.
7. Type of an actual parameter in a function call differs from type previously encountered or defined in that position.
8. Function definition within a function definition.
9. Function defined twice.
10. Number of formal parameters in a function definition does not match the number of actual parameters previously used in calls for the function.
11. Type of a formal parameter in a function definition does not match type previously encountered in that position in function calls.
12. Jump to an undefined line number.
13. Jump into a function definition.
14. Jump out of a function definition.
15. Function name used as a destination outside the definition of that function.
16. Next statement does not have corresponding For statement.
17. For statement does not have corresponding Next statement.
18. Function called is not defined.
19. FN END statement is missing.
20. Formal parameters in a formal parameter list are identical.

3. ALGOL60

In the case of ALGOL60 the syntax rules for the language were clearly defined in BNF notation by Naur, *et al.* (1963). However, the static semantic rules were expressed rather vaguely as a set of English sentences. As a result there are situations where the validity of an ALGOL program cannot be determined from the ALGOL60 report. This makes the problem of implementation more difficult for the compiler writer and the problem of standardisation becomes impossible.

The specification given here defines the static semantic rules completely. In cases where it was not clear from the original report what the outcome should be, a decision was taken one way or another. The specification again consists of two passes and requires in addition a number of run-time routines for dynamic checking of parameter types.

3.1 *Syntax rules for ALGOL60*

1. < expression > ::= < arith exp > | < bool exp > | < des exp >
2. < variable id > ::= < id >
3. < simple var > ::= < variable id >
4. < subs exp > ::= < arith exp >
5. < subs list > ::= < subs exp > | < subs list >, < sub exp >
6. < array id > ::= < id >
7. < array ref > ::= < id >
8. < subs var > ::= < array ref > [< subs list >]
9. < var > ::= < subs var > | < simple var >
10. < proc id > ::= < id >
11. < function id > ::= < id >
12. < actual par > ::= < string > | < id > | < expression >
13. < letter string > ::= < letter > | < letter string > < letter >
14. < par delimiter > ::= , |) < letter string > :(
15. < actual par list > ::= < actual par > |
< actual par list > < par delimiter > < actual par >
16. < actual par part > ::= < empty > |(< actual par list >)
17. < function des > ::= < function id > < actual par part >
18. < adding op > ::= + | −
19. < primary > ::= < unsigned integer no > |
< unsigned real no > | < var > | < function des > |
(< arith exp >)
20. < factor > ::= < primary > | < factor > $\uparrow$ < primary >
21. < term > ::= < factor > | < term > $\times$ < factor > |
< term > / < factor > | < term > $\div$ < factor >
22. < simple a e > ::= < term > | < adding op > < term > |
< simple a e > < adding op > < term >
23. < if clause > ::= **if** < bool exp > **then**
24. < arith exp > ::= < simple a e > |
< if clause > < simple a e > **else** < arith exp >
25. < relational op > ::= < | $\leqslant$ | = | $\geqslant$ | > | $\neq$
26. < relation > ::=
< simple a e > < relational op > < simple a e >
27. < boolean prim > ::= < logical value > | < var > |
< function des > | < relation > |(< bool exp >)
28. < boolean sec > ::= < boolean prim > |
$\neg$ < boolean prim >
29. < boolean factor > ::= < boolean sec > |
< boolean factor > $\wedge$ < boolean sec >
30. < boolean term > ::= < boolean factor > |
< boolean ter > $\vee$ < boolean factor >
31. < implication > ::= < boolean term > |
< implication > $\supset$ < boolean term >
32. < simple boolean > ::= < implication > |
< simple boolean > $\equiv$ < implication >
33. < bool exp > ::= < simple boolean > |
< if clause > < simple boolean > **else** < bool exp >
34. < label > ::= < id > | < unsigned integer >
35. < label ref > ::= < id > | < unsigned integer >
36. < switch id > ::= < id >
37. < switch ref > ::= < id >
38. < switch des > ::= < switch ref > [< subs exp >]
39. < simple d e > ::= < label ref > | < switch des > |
(< des exp >)
40. < des exp > ::= < simple d e > |
< if clause > < simple d e > **else** < des exp >
41. < unlabelled basic stm > ::= < ass st > | < goto st > |
< dummy st > | < proc st >
42. < basic stm > ::= < unlabelled basic stm > |
< label > : < basic stm >
43. < unconditional stm > ::= < basic stm > |
< compound stm > | < block >
44. < stm > ::= < unconditional stm > | < conditional stm > |
< for st >
45. < compound tail > ::= < stm > **end** |
< stm > ; < compound tail >
46. < begin symbol > ::= **begin**

47. < block head > ::= < begin symbol > < declaration > |
< block head > ; < declaration >
48. < unlabelled compound > ::= **begin** < compound tail >
49. < unlabelled block > ::=
< block head > ; < compound tail >
50. < compound stm > ::= < unlabelled compound > |
< label > : < compound stm >
51. < block > ::= < unlabelled block > | < label > : < block >
52. < program > ::= < block > | < compound stm >
53. < left part > ::= < subs var > := | < id > :=
54. < left part list > ::= < left part > |
< left part list > < left part >
55. < ass st > ::= < left part list > < arith exp > |
< left part list > < bool exp >
56. < goto st > ::= **goto** < des exp >
57. < dummy st > ::= < empty >
58. < if st > ::= < if clause > < unconditional stm >
59. < conditional stm > ::= < if st > | < if st > **else** < stm > |
< if clause > < for st > | < label > : < conditional stm >
60. < for list element > ::= < arith exp > |
< arith exp > **step** < arith exp > **until** < arith exp > |
< arith exp > **while** < bool exp >
61. < for list > ::= < for list element > | < for list >, < for list element >
62. < for clause > ::= **for** < var > := < for list > **do**
63. < for st > ::= < for clause > < stm > | < label > : < for st >
64. < proc st > ::= < proc id > < actual par part >
65. < declaration > ::= < type decl > | < array decl > |
< switch decl > | < proc decl >
66. < type list > ::= < simple var > |
< simple var >, < typelist >
67. < type > ::= **real** | **integer** | **Boolean**
68. < local or own type > ::= < type > | **own** < type >
69. < type decl > ::= < local or own type > < type list >
70. < lower bound > ::= < arith exp >
71. < upper bound > ::= < arith exp >
72. < bound pair > ::= < lower bound > : < upper bound >
73. < bound pair list > ::= < bound pair > |
< bound pair list >, < bound pair >
74. < array seg > ::= < array id > [< bound pair list >] |
< array id >, < array seg >
75. < array list > ::= < array seg > |
< array list >, < array seg >
76. < array specifier > ::= **array** | < local or own type > **array**
77. < array decl > ::= < array specifier > < array list >
78. < switch list > ::= < des exp > | < switch list >, < des exp >
79. < switch decl > ::= **switch** < switch id > := < switch list >
80. < formal par > ::= < id >
81. < formal par list > ::= < formal par > |
< formal par list > < par delimiter > < formal par >
82. < formal par part > ::= < empty > | (< formal par list >)
83. < id list > ::= < id > | < id list >, < id >
84. < value part > ::= < empty > | **value** < id list > ;
85. < specifier > ::= **string** | < type > | **array** | < type > **array** |
label | **switch** | **procedure** | < type > **procedure**
86. < spec part > ::= < empty > | < specifier > < id list > ; |
< spec part > < specifier > < id list > ;
87. < proc name > ::= < id >
88. < proc heading > ::= < proc name > < formal par part > ;
< value part > < spec part >
89. < proc body > ::= < stm > | < code >
90. < proc decl > ::=
procedure < proc heading > < proc body > |
< type > **procedure** < proc heading > < proc body >

3.2 *Static semantic rules for pass I*
The first pass is concerned with the construction of a symbol table with entries for each identifier in the program. This is done using:

(*a*) ICBS (Identifiers in Current Block Stack) which has entries of form: (identifier, type, structure, actual/formal, n, x, y)
where type = 'R', 'I', 'B' or 0,
structure = 'V' (variable), 'A' (array), 'F' (function), 'P' (procedure), 'S' (switch), 'L' (label) or 'STR' (string),
actual/formal = 'A' (actual variable) or 'F' (formal parameter),
n = number of dimensions (array) or parameters (procedure or function),
x = position of formal parameter in formal parameter list, and
y = name of procedure or function with formal parameter list;
(*b*) DUMP which stores the contents of the ICBS when entering an inner block;
(*c*) BS (Block Stack) which has entries of form:
(No of level, ICBS for this level)
(*d*) NBS (Nested Block Stack) which contains level numbers of each incomplete (nested) block; and
(*e*) SS (Symbol Stack) which contains the ICBS for each level in order.

Other stacks used are:

(*a*) T	(Type stack) contains entries of form: type;
(*b*) SL	(Subscript List) contains count of number of subscripts;
(*c*) L	is a stack of identifiers (id or array id);
(*d*) FPL	(Formal Parameter List) is a stack of formal parameter identifiers;
(*e*) PNL	(Procedure Name List) contains the current procedure (or function) name;
(*f*) VL	(Value List) contains identifiers called by value;
(*g*) S	(Structure stack) contains entries of form: structure ('V', 'A', etc.).

34. < label > ::= < id > {𝒮(ICBS, < id >, E2, (< id >, 0, 'L', 'A', 0,0,0)↓ICBS)} |
< unsigned integer > {𝒮(ICBS, < unsigned integer >, E2, (< unsigned integer >, 0, 'L', 'A', 0,0,0)↓ICBS)}
67. < type > ::= **real** {'R'↓T} | **integer** {'I'↓T} | **Boolean** {'B'↓T}
66. < type list > ::= < simple var > {𝒮(ICBS, < simple var >, E1, (< simple var >, 𝒯(T), 'V', 'A', 0,0,0)↓ICBS)} |
< simple var >, < type list > {**ditto**}
69. < type decl > ::= < local or own type > < type list > {λ↑T}
73. < bound pair list > ::= < bound pair > {'1'↓SL} |
< bound pair list >, < bound pair > {**ditto**}
74. < array seg > ::= < array id > [< bound pair list >]
{< array id > ↓L; ∀L:(y↑L; 𝒮(ICBS, y, E1, (y, 𝒯(T), 'A', 'A', 𝒩(SL), 0,0)↓ICBS)); λ ⇑ SL} |
< array id >, < array seg > { < array id > ↓L}
76. < array specifier > ::= **array** {'R'↓T} |
< local or own type > **array**
77. < array decl > ::= < array specifier > < array list > {λ↑T}
79. < switch decl > ::= **switch** < switch id > := < switch list >
{𝒮(ICBS, < switch id >, E1, (< switch id >, 0, 'S', 'A', 0,0,0)↓ ICBS)}
87. < proc name > ::= < id > {𝒮(ICBS, < id >, E1, < id > ↓ PNL)}
80. < formal par > ::= < id > {𝒮(FPL, < id >, E3, (< id >, 0, 'FP', 'F', 0, 𝒩(FPL), 𝒯(PNL))↓FPL)}
83. < id list > ::= < id > { < id > ↓L} | < id list >, < id > {**ditto**}
84. < value part > ::= < empty > | **value** < id list > ; {∀L:(x↑L; 𝒮(VL, x, E4, x↓VL))}
85. < specifier > ::= **string** {'STR'↓S; 0↓T} | < type > {'V'↓S} |
array {'A'↓S; 'R'↓T} | < type > **array** {'A'↓S} |
label {'L'↓S; 0↓T} | **switch** {'S'↓S; 0↓T} |
procedure {'P'↓S; 0↓T} | < type > **procedure** {'P'↓S}

86. < spec part > ::= < empty > | < specifier > < id list > ; {y↑T;z↑S;∀L:(x↑L;'F'→u; **if** z='V' **or** 'A' **or** 'L' **then** 𝒮(VL,x,λ→VL;'V'→u, −);𝒮(FPL,x, **if** Comp3≠'FP' **then** E5 **else** (y→Comp2;z→Comp3;u→Comp4),E6))} |
< spec part > < specifier > < id list > ; {**ditto**}

88. < proc heading > ::= < proc name > < formal par part > ; < value part > < spec part > {y↑PNL;x↑T;
if 𝒩(T)=0 **then** (x↓T;(y,0,'P','A',𝒩(FPL),0,0)↓ICBS) **else** (y,x,'F', 'A',𝒩(FPL),0,0)↓ICBS; DUMP ⇑ ICBS; 𝒩(BS)↓NBS; (𝒩(BS),0)↓BS;x ⇑ FPL;x ⇓ ICBS; ∀VL:(x↑VL; **if** x≠λ**then** E7)}

90. < proc decl > ::= **procedure** < proc heading > < proc body > {x↑NBS;y ⇑ ICBS;𝒮(BS,x,y→Comp2, −); DUMP ⇓ ICBS;λ↑DUMP} |
< type > **procedure** < proc heading > < proc body > {**ditto**}

46. < begin symbol > ::= **begin** {DUMP ⇑ ICBS;𝒩(BS)↓NBS; (𝒩(BS),0)↓BS}

49. < unlabelled block > ::= < block head > ; < compound tail > {x↑NBS;y ⇑ ICBS;𝒮(BS,x,y→Comp2, −); DUMP ⇓ ICBS;λ↑DUMP}

52. < program > ::= < block > {∀BS:(x↑BS;Comp2(x)↓SS)} |
< compound stm > {**ditto**}

3.3 *Static semantic rules for pass II*

The result of the first pass is a complete symbol table SS containing the identifiers declared at each level. This is retained for use in the second pass. The second pass checks the uses of identifiers (arrays, procedures, formal parameters, labels, etc.) against their definitions to ensure that each identifier is used correctly.

Stacks used in the second pass are the following:

(*a*) ST (Symbol Table) which contains symbol table entries from SS for current blocks

(*b*) SL (Subscript List) which counts the number of subscripts in the subscript list of a subscripted variable

(*c*) APL (Actual Parameter List) which counts the number of actual parameters in a procedure or function call

(*d*) DUMP which stores SL and APL when nesting occurs

(*e*) STDUMP which stores one level of symbol table entries while evaluating dynamic array bound expressions in an array declaration

(*f*) FASL (Function Assignment Statement List) which stores the name of each function while it is being defined, together with a marker to indicate whether it contains an assignment statement with the function name as left part

(*g*) L which is used as a temporary stack

(*h*) TS (Type Stack) which has entries of form
(type, structure, n, x, y, APL, formal par stack)

where type	=	'R', 'I', 'B', 'FP', 'FPS' or 0 (if structure = 'L')
structure	=	'VE', 'V', 'A', 'P' or 'L',
n	=	number of dimensions or parameters,
x	=	position in formal parameter list,
y	=	name of procedure (function) to which formal parameter 'belongs',
APL	=	actual parameter list (in case of function call)

and formal par stack = stack of all formal parameters whose type is not determined at any point in an expression; this has entries of form:

(structure, n, x, y, APL)

(*i*) LPS (Left Part Stack) which has entries of the same form as TS

(*j*) RTFPS (Run Time Formal Parameter Stack) which has entries of form:

(procedure or function name, stack)

where stack is a stack of entries (one for each formal parameter) of form:

(type, structure, value)

where type	=	'R', 'I', 'B' or 0,
structure	=	'V', 'A', 'F', 'P', 'S', 'STR', 'L' or 'FP',
and value	=	'V' or 0.

When an identifier is encountered in an expression an entry is placed on the Type Stack. If the identifier is not a formal parameter, or if it is a formal parameter whose type is specified, the Type Stack entry will have 'R', 'I' or 'B' in Component 1 and zeros in all other components. If the identifier is a formal parameter of unspecified type, the entry has 'FP' as first component and non-zero Components 4 and 5. From the context in which the formal parameter is used, parameters 2 and 3 are deduced.

When a production is encountered which determines the type of its dependent formal parameter(s), the appropriate Run Time Check(s) can be inserted into the object code and the type stack entry removed or replaced by one with 'R', 'I' or 'B' in Component 1 and zeros in the other components. If the two top items of TS are both of type 'FP' and the production cannot deduce what type either should be, the two items are replaced by a single entry which has 'FPS' as Component 1, zeros in Components 2 to 6 and Components 2 to 6 of the original two items are placed in the stack in Component 7 of the resulting item.

The specification for pass II is as follows:

1. < expression > ::= < arith exp > | < bool exp > {('B',0,0,0, 0,0,0)↓TS} | < des exp > {(0,'L',0,0,0,0,0)↓TS}

5. < subs list > ::= < subs exp > {'1'↓SL} | < subs list > , < subs exp > {**ditto**}

7. < array ref > ::= < id > {DUMP ⇑ SL}

8. < subs var > ::= < array ref > [< subs list >] {𝒮(ST, < array ref > , **if** Comp3='FP' **then** ('FP','A', 𝒩(SL),Comp6,Comp7,0,0)↓TS **else if** Comp3≠'A' **then** E15 **else** ((Comp2,0,0,0,0,0,0)↓TS; **if** Comp4='A' **then** (**if** Comp5≠𝒩(SL) **then** E30) **else** 'Rtcheck(Comp6, Comp7, Comp2,'A', 𝒩(SL))'),E10);λ ⇑ SL;DUMP ⇓ SL; λ↑DUMP}

9. < var > ::= < subs var > | < simple var > {𝒮(ST, < simple var > , **if** Comp3='FP' **then** ('FP', 'VE', 0,Comp6,Comp7,0,0)↓TS **else if** Comp3='V' **then** (**if** Comp4='F' **then** (Comp2,'VE',0,Comp6,Comp7,0,0)↓TS **else** (Comp2,0,0,0,0,0,0)↓TS) **else** E14,E9)}

19. < primary > ::= < unsigned integer no > {('I',0,0,0,0,0,0) ↓TS} |
< unsigned real no > {('R',0,0,0,0,0,0)↓TS} |
< var > {𝒯(TS)→x; **if** Comp1(x)='B' **then** E21} |
< function des > {𝒯(TS)→x; **if** Comp1(x)='B' **then** E22} |
(< arith exp >)

20. < factor > ::= < primary > | < factor > ↑ < primary > {①}

21. < term > ::= < factor > | < term > × < factor > {①} |
< term > / < factor > {x↑TS;②;x↑TS;②;('R',0,0,0,0,0,0) ↓TS} |
< term > ÷ < factor > {x↑TS; **if** Comp1(x)='R' **then** E23 **else** ③;x↑TS; **if** Comp1(x)='R' **then** E23 **else** ③; ('I',0,0,0,0,0,0)↓TS}

22. < simple a e > ::= < term > | < adding op > < term > |
< simple a e > < adding op > < term > {①}

24. < arith exp > ::= < simple a e > |
< ifclause > < simple a e > **else** < arith exp > {①}

26. < relation > ::= < simple a e > < relational op > < simple a e > {x↑TS;②;x↑TS;②}

27. < boolean prim > ::= < logical value > | < var > {x↑TS; **if** Comp1(x)='R' **or** 'I' **then** E28 **else if** Comp1(x)='FP' **then** 'Rtcheck(Comp4(x),Comp5(x),'B',Comp2(x), Comp3(x))'} |
< function des > {x↑TS; **if** Comp1(x)='R' **or** 'I' **then** E29 **else if** Comp1(x)='FP' **then** ('Rtcheck(Comp4(x),

Comp5(x),'B',Comp2(x),Comp3(x))';
'Rtparlistcheck(Comp4(x),Comp5(x),Comp6(x))')} |
< relation > |(< bool exp >)

53. < left part > ::= < subs var > := {LPS↑TS} |
< id > := {𝒮(ST, < id >, **if** Comp3 = 'V' ∧ (Comp4 = 'A' **or** 'V') **then** (Comp2,0,0,0,0,0,0)↓LPS **else if** Comp3 = 'F' ∧ Comp4 = 'A' **then** ((Comp2,0,0,0,0,0,0) ↓LPS;𝒮(FASL, < id >,'1'→Comp2, E32)) **else if** Comp3 = 'V' ∧ Comp4 = 'F' **then** ((Comp2,0,0,0,0,0,0) ↓LPS; 'Rtcheck(Comp6,Comp7,Comp2,'V',0)') **else if** Comp3 = 'FP' **then** ('FP','V',0,Comp6,Comp7,0,0)↓LPS **else** E20, E19)}

54. < left part list > ::= < left part > |
< left part list > < left part >
{x↑LPS;y↑LPS; **if** Comp1(y) = 'R' **then** (y↓LPS; **if** Comp1(x) = 'I' **then** E24 **else if** Comp1(x) = 'FP' **then** 'Rtcheck(Comp4(x),Comp5(x),'R',Comp2(x),Comp3(x))' **else if** Comp1(x) = 'FPS' **then** (Comp7(x)↓L;∀L:(y↑L; 'Rtcheck(Comp3(y),Comp4(y),'R',Comp1(y), Comp2(y))'))) **else if** Comp1(x) = 'R' **then** (x↓LPS; **if** Comp1(y) = 'I' **then** E24 **else if** Comp1(y) = 'FP' **then** 'Rtcheck(Comp4(y),Comp5(y),'R',Comp2(y),Comp3(y))' **else if** Comp1(y) = 'FPS' **then** (Comp7(y)↓L;∀L:(x↑L; 'Rtcheck(Comp3(x),Comp4(x),'R',Comp1(x), Comp2(x))'))) **else if** Comp1(y) = 'I' **then** (y↓LPS; **if** Comp1(x) = 'FP' **then** 'Rtcheck(Comp4(x),Comp5(x),'I', Comp2(x),Comp3(x))' **else if** Comp1(x) = 'FPS' **then** (Comp7(x)↓L;∀L:(y↑L;'Rtcheck(Comp3(y),Comp4(y),'I', Comp1(y),Comp2(y))')))) **else if** Comp1(x) = 'I' **then** (x↓LPS; **if** Comp1(y) = 'FP' **then** 'Rtcheck(Comp4(y), Comp5(y),'I',Comp2(y),Comp3(y))' **else if** Comp1(y) = 'FPS' **then** (Comp7(y)↓L;∀L:(x↑L;'Rtcheck (Comp3(x), Comp4(x),'I',Comp1(x),Comp2(x))'))) **else** (**if** Comp1(y) = 'FP' **then** (Comp2(y),Comp3(y),Comp4(y), Comp5(y),Comp6(y))↓L **else** Comp7(y) ⇓ L; (Comp2(x), Comp3(x),Comp4(x),Comp5(x),Comp6(x))↓L;x ⇑ L; ('FPS',0,0,0,0,0,x)↓LPS)}

55. < ass st > ::= < left part list > < arith exp > {x↑TS;②; x↑LPS; **if** Comp1(x) = 'B' **then** E25; **if** Comp1(x) = 'FP' **then** 'Rtcheck(Comp4(x),Comp5(x),'RI',Comp2(x), Comp3(x))' **else if** Comp1(x) = 'FPS' **then** 'Rtchecklist (Comp7(x))'} |
< left part list > < bool exp > {x↑LPS; **if** ,Comp1(x) = 'R' **or** 'I' **then** E26; **if** Comp1(x) = 'FP' **then** ('Rtcheck (Comp4(x),Comp5(x),'B',Comp2(x),Comp3(x))'; **if** Comp2(x) = 'F' **then** 'Rtparlistcheck(Comp4(x), Comp5(x),Comp6(x))') **else if** Comp1(x) = 'FPS' **then** (Comp6(x)↓L;∀L:(y↑L;'Rtcheck(Comp3(y),Comp4(y), 'B',Comp1(y),Comp2(y))'; **if** Comp1(y) = 'F' **then** 'Rtparlistcheck(Comp3(y),Comp4(y),Comp5(y))'))}

60. < for list element > ::= < arith exp > {x↑TS;②} |
< arith exp > **step** < arith exp > **until** < arith exp > {x↑TS; ②;x↑TS;②;x↑TS;②} |
< arith exp > **while** < bool exp > {x↑TS;②}

62. < for clause > ::= **for** < var > := < for list > **do** {x↑TS; **if** Comp1(x) = 'B' **then** E27 **else if** Comp1(x) = 'FP' **then** (**if** Comp2(x) = 'VE' **then** 'V'→Comp2(x); 'Rtcheck(Comp4(x),Comp5(x),'R',Comp2(x),Comp3(x))') **else if** Comp2(x) = 'VE' **then** 'Rtcheck(Comp4(x), Comp5(x),Comp1(x),'V',Comp3(x))'}

70. < lower bound > ::= < arith exp > {x↑TS;②}

71. < upper bound > ::= < arith exp > {x↑TS;②}

4. < subs exp > ::= < arith exp > {x↑TS;②}

10. < proc id > ::= < id > {DUMP ⇑ APL}

11. < function id > ::= < id > {DUMP ⇑ APL}

12. < actual par > ::= < string > {(0,'STR',0,0,0,0)↓APL} |
< expression > {x↑TS; **if** Comp1(x) = 'FP' **then** (0, 'VE', (Comp4(x),Comp5(x),Comp2(x),Comp3(x)),0,0,0)↓APL **else if** Comp1(x) = 'FPS' **then** (0,'VE',Comp7(x),0,0,0) ↓APL **else if** Comp2(x) ≠ 'L' **then** (Comp1(x),'VE',0,0,0,0) ↓APL **else** (0,'L',0,0,0,0)↓APL} |
< id > {𝒮(ST, < id >, **if** Comp3 = 'F' ∧ Comp4 = 'A' **then** (Comp2,Comp3,Comp5,0,0, < id >)↓APL **else** (Comp2,Comp3,Comp5,Comp6,Comp7,0)↓APL, E18)}

17. < function des > ::= < function id > < actual par part > {𝒩(APL)→y;x ⇑ APL;𝒮(ST, < function id >, **if** Comp3 = 'FP' **then** ('FP','F',y,Comp6,Comp7,x,0)↓TS **else if** Comp3 ≠ 'F' **then** E17 **else** ((Comp2,0,0,0,0,0,0) ↓TS; **if** Comp4 = 'A' **then** (**if** Comp5 ≠ y **then** E31; 'Rtparlistcheck(< function id > ,0,x)') **else** ('Rtcheck(Comp6,Comp7,Comp2,'F',y)'; 'Rtparlistcheck(Comp6,Comp7,x)')),E12);DUMP ⇓ APL; λ↑DUMP}

64. < proc st > ::= < proc id > < actual par part > {𝒮(ST, < proc id >, **if** Comp3 ≠ 'FP' **and** 'P' **then** E17; **if** Comp4 = 'A' **then** (**if** Comp5 ≠ 𝒩(APL) **then** E31; 'Rtparlistcheck(< proc id > ,0,APL)') **else** ('Rtcheck (Comp6,Comp7,0,'P',𝒩(APL))'; 'Rtparlistcheck(Comp6,Comp7,APL)'), E12);λ ⇑ APL; DUMP ⇓ APL;λ↑DUMP}

80. < formal par > ::= < id > {𝒮(ST, < id >, **if** Comp4 = 'V **then** 'V'→x **else** 0→x; (Comp2,Comp3,x)↓FPL, –)}

88. < proc heading > ::= < proc name > < formal par part > ; < value part > < spec part > {𝒮(ST, < proc name >, **if** Comp3 = 'F' **then** (< proc name > ,0)↓FASL, –);y ⇑ FPL; (< proc name > ,y)↓RTFPS}

90. < proc decl > ::=
procedure < proc heading > < proc body > {λ↑ST} |
< type > **procedure** < proc heading > < proc body > {λ↑ST;x↑FASL; **if** Comp2(x) = 0 **then** E33}

35. < label ref > ::= < id > {𝒮(ST, < id >, **if** Comp3 ≠ 'L' **and** 'FP' **then** E13, E8)} |
< unsigned integer > {𝒮(ST, < unsigned integer >, **if** Comp3 ≠ 'L' **then** E13, E8)}

37. < switch ref > ::= < id > {𝒮(ST, < id >, **if** Comp3 ≠ 'S' **and** 'FP' **then** E16, E11)}

46. < begin symbol > ::= **begin** {ST↑SS}

49. < unlabelled block > ::= < block head > ; < compound tail > {λ↑ST}

76. < array specifier > ::= **array** {STDUMP↑ST} |
< local or own type > **array** {**ditto**}

77. < array decl > ::= < array specifier > < array list > {ST↑STDUMP}

where

① = x↑TS; y↑TS; **if** Comp1(y) = 'R' **then** (y↓TS; ②) **else if** Comp1(x) = 'R' **then** (x↓TS; y→x;②) **else**
if Comp1(y) = 'I' **then** x↓TS **else if** Comp1(x) = 'I' **then** y↓TS **else**
if Comp1(y) = 'FP' **then** ((Comp2(y),Comp3(y), Comp4(y),Comp5(y),Comp6(y))↓L;
if Comp1(x) = 'FP' **then** (Comp2(x),Comp3(x), Comp4(x),Comp5(x),Comp6(x))↓L
else Comp7(x) ⇓ L;x ⇑ L; ('FPS',0,0,0,0,0,x)↓TS) **else**
if Comp1(x) = 'FP' **then** ((Comp2(x),Comp3(x), Comp4(x),Comp5(x),Comp6(x))↓L;
Comp7(y) ⇓ L;x ⇑ L;('FPS',0,0,0,0,0,x)↓TS) **else**
(Comp7(x) ⇓ L;Comp7(y) ⇓ L;x ⇑ L;('FPS',0,0,0,0,0,x) ↓TS)

② = **if** Comp1(x) = 'FP' **then** ('Rtcheck(Comp4(x),Comp5(x), 'RI',Comp2(x),Comp3(x))';
if Comp2(x) = 'F' **then** 'Rtparlistcheck(Comp4(x), Comp5(x),Comp6(x))') **else**
if Comp1(x) = 'FPS' **then** (Comp7(x)↓L; ∀L: (y↑L; 'Rtcheck(Comp3(y),Comp4(y),'RI',Comp1(y), Comp2(y))'; **if** Comp1(y) = 'F' **then** ' Rtparlistcheck (Comp3(y),Comp4(y),Comp5(y))'))

③ = **if** Comp1(x) = 'FP' **then** ('Rtcheck(Comp4(x),Comp5(x),

```
    'I',Comp2(x),Comp3(x))'; if Comp2(x)='F' then
    'Rtparlistcheck(Comp4(x),Comp5(x),Comp6(x))') else
    if Comp1(x)='FPS' then (Comp7(x)↓L;∀L:(y↑L;
    'Rtcheck(Comp3(y),Comp4(y),'I',Comp1(y),
    Comp2(y))'; if Comp1(y)='F' then 'Rtparlistcheck
    (Comp3(y),Comp4(y),Comp5(y))'))
```

3.4 *Run time routines*

The format of these is not defined by the notation; however, the action performed by them can be expressed by means of the notation. Two stacks are used: RTFPS (Run Time Formal Parameter Stack) which is set up in Pass II, and RTAPS (Run Time Actual Parameter Stack) which is maintained by the run time routines.

```
Rtparlistcheck (i, j, APL) =
    {if j=0 then i→k else (𝒮(RTAPS,j,Comp2 ⇓ Z,−);
    𝒮(Z,i,Comp5→k,−);λ ⇑ Z);𝒮(RTFPS,k,Comp2↓X,−);
    ∀X:(x↑X;y↑APL; if Comp2(y)='FP' then 𝒮(RTAPS,
    Comp5(y),Comp2 ⇓ Z,−);𝒮(Z,Comp4(y),Comp2→
    Comp1(x);Comp3→Comp2(x);Comp4→Comp3(x);
    Comp5→Comp6(x),−);λ ⇑ Z); if Comp2(x)='FP' then
    (𝒩(X),Comp1(y),Comp2(y),Comp3(y),Comp6(y))↓PARS
    else if Comp2(y)≠'VE' then (if Comp1(y)=Comp1(x) ∧
    Comp2(y)=Comp2(x) then (𝒩(X),Comp1(y),Comp2(y),
    Comp3(y),Comp6(y))↓PARS else E34) else (if Comp1(y)=0
    then (Comp3(y) ⇓ Z;∀Z:(z↑Z;'Rtcheck(Comp1(z),Comp2(z),
    Comp1(x),Comp3(z),Comp4(z))'); if Comp2(x)≠'V' then
    E35) else if Comp1(y)≠Comp1(x) ∨ Comp2(x)≠'V' then
    E35; if Comp3(x)='V' then (𝒩(X),Comp1(x),'V',0,0)↓PARS
    else (𝒩(X),Comp1(x),'VE',0,0)↓PARS)); x ⇑ PARS;
    (k,x)↓RTAPS}
```

```
Rtcheck(x,y,i,j,k)=
    {𝒮(RTAPS,y,Comp2↓S,−);𝒮(S,x, if Comp3≠'VE' ∨
    Comp2≠0 then (if (i≠'RI' ∨ Comp2≠'R' and 'I') ∧
    Comp2≠i then E36 else if Comp3≠j ∨ Comp4≠k then E37)
    else (if j≠'VE' then E38; Comp4 ⇓ TEMP; ∀TEMP:
    (t↑TEMP; 'Rtcheck(Comp1(t),Comp2(t),i, Comp3(t),
    Comp4(t))')),−);λ ⇑ S}
```

```
Rtchecklist (i) =
    {𝒯(i)→x;𝒮(RTAPS,Comp4(x),Comp2(x)↓S,−);
    𝒮(S,Comp3(x),Comp2→y,−); if y≠'R' and 'I' then E20;
    λ ⇑ S;∀i:(x↑i;𝒮(RTAPS,Comp4(x),Comp2↓S,−);
    𝒮(S,Comp3(x), if Comp3='VE' ∧ Comp2=0 then E 38
    else if Comp2≠y then E39 else if Comp3≠Comp1(x) ∨
    Comp4≠Comp2(x) then E40,−);λ ⇑ S)}
```

3.5 *Error numbers*

1. Identifier is declared more than once in the same block.
2. Symbol used as label has already been declared in the same block.
3. Formal parameter identifier appears more than once in the same formal parameter list.
4. Formal parameter appears more than once in the same *value* list.
5. Formal parameter appears more than once in the specification part of a procedure declaration.
6. Identifier appears in the specification part of a procedure declaration which does not occur in the formal parameter list.
7. One or more identifiers in the value list of a procedure declaration do not correspond to formal parameters specified to be simple variables, arrays or labels.
8. Label referred to in a designationial expression does not exist.
9. Simple variable referred to in an expression has not been declared.
10. Array name referred to has not been declared.
11. Switch identifier referred to has not been declared.
12. Procedure identifier referred to has not been declared.
13. Label referred to in a designationial expression has been declared to have structure different from label.
14. Simple variable referred to in an expression has been declared to have structure different from simple variable.
15. Array name referred to has been declared to have structure different from array.
16. Switch identifier referred to has been declared to have structure different from switch.
17. Procedure identifier referred to has been declared to have structure different from procedure.
18. Identifier used as an actual parameter has not been declared.
19. Identifier appearing as a left part has not been declared.
20. Identifier appearing as a left part has incorrect structure.
21. Boolean variable used as a primary in an arithmetic expression.
22. Boolean procedure called in an arithmetic expression.
23. At least one of the operands of the operator ÷ is not of type integer.
24. Types of identifiers appearing in the left part list of an assignment statement are different.
25. An arithmetic expression assigned to a Boolean variable.
26. A Boolean expression assigned to an arithmetic variable.
27. A Boolean variable used as the controlled variable of a For statement.
28. An arithmetic variable used as a Boolean primary in a Boolean expression.
29. An arithmetic procedure used as a Boolean primary in a Boolean expression.
30. The number of subscripts in a subscripted variable does not match the number of dimensions of the array in the array declaration.
31. The number of actual parameters in a procedure call does not match the number of formal parameters in the procedure declaration.
32. Assignment to a function identifier outside the body of the function.
33. Function does not contain an assignment statement with the function identifier as left-part variable.
34. The type/structure of an identifier used as an actual parameter does not match the type/structure of the corresponding formal parameter as declared in the specification part.
35. The type/structure of an expression used as an actual parameter does not match the type/structure of the corresponding formal parameter as declared in the specification part.
36. The type of an identifier used as a formal parameter does not match the type required by its use.
37. The structure/number of dimensions or parameters of an actual parameter does not match the structure/number of dimensions or parameters of the formal parameter as it is used.
38. An expression is used as an actual parameter whereas the corresponding formal parameter expects an identifier (e.g. array id).
39. An actual parameter which is substituted for a formal parameter used as a left-part variable of an assignment statement, does not match the types of the other parameters substituted as left-part variables.
40. An actual parameter which is substituted for a formal parameter used as a left-part variable of an assginment statement, does not match the structure/number of dimensions of its use.
41. The type of a formal parameter used in an actual parameter expression is causing a mismatch with the declared type (in the specification part) of the formal parameter for which the

actual parameter is to be substituted.

42. The structure/number of dimensions or parameters of a formal parameter used in an actual parameter expression does not match its use.
43. A formal parameter used in an actual parameter expression expects an identifier to be substituted for it, but finds instead an expression.

4. Conclusion

This study demonstrates how the static semantic rules of two common programming languages can be specified formally. The static semantic rules of ALGOL60 are much more difficult to specify than those of BASIC, partly because of the ability to mix types within an expression under certain circumstances but not under others (e.g. ÷), and partly because the type and structure of a formal parameter need not be declared.

For the complete specification of ALGOL60, situations were encountered for which the correctness could not be decided from the original report. The implementation of such situations is left to the compiler writer's discretion. However, if one is to standardise programming languages so that a program written in a given high level language has the same syntax and produces the same effect (wherever possible) when run on different machines, one needs to specify all aspects of the language completely. Thus in the case of undecideable situations in ALGOL60 a decision was taken one way or another. Thus the specification of ALGOL60 given here will produce a similar effect to that produced by some ALGOL60 compilers but not by others. It is not suggested that the specification given here should necessarily be the generally accepted one; however, the intention is to show that the complex static semantic rules of ALGOL60 can be expressed in a formal notation which is meaningful to compiler writers and hence assist in the standardisation of such a language.

Besides its effect on standardisation such a formal specification may also be of assistance to the compiler writer from the point of view of the correctness of the compiler he produces by giving him a model of how these aspects of the language can be implemented at the very simplest level. Obviously the compiler writer will have his own ideas as to how best to implement certain features to provide the most efficient system. However, at a time when the correctness of the product is as important as its efficiency, by providing the compiler writer with a simply implementable specification of these aspects of the language, he has a yardstick by which to measure the correctness of his own implementation.

References

Bull, G. M., Freeman, W. and Garland, S. J. (1973). *Specification for Standard BASIC*, NCC Publications: Manchester.
Ledgard, H. F. (1969). A Formal System for Defining the Syntax and Semantics of Computer Languages, Ph.D. Thesis, MIT.
Lee, J. A. N. (1972). The Formal Definition of the BASIC Language, *The Computer Journal*, vol. 15, pp. 37-41.
Naur, P. *et al.* (1963). Revised report on the algorithmic language ALGOL60, *The Computer Journal*, vol. 5, pp. 349-367.
Williams, M. H. (1978). A Formal Notation for specifying Static Semantic Rules, submitted to *Computer Languages.*

Book reviews

The Complexity of Computational Problem Solving, edited by R. S. Anderssen and R. P. Brent, 1976; 262 pages. (*University of Queensland Press*, £4·70)

The title is mildly ambiguous, since this book is not so much concerned with problem solving in the usual sense (although this is certainly involved) as complexity in all its computational aspects, and understood indeed in a rather broad way to include efficiency in the formulation, programming and debugging phases of program construction. The problems are therefore primarily with respect to the methods of computing and less with the 'external' problem for which the computer is to be used.

An example of an important external problem is the last of the fifteen articles which is called 'the complexity of real-world scheduling problems'. These problems include resource allocation and assignment, timetabling, job/shop scheduling and the like. These are so complicated that they require overconstrained linear programs which are heuristic in nature. The extent to which such programs depart from optimality can also be measured.

Three papers in the book concentrate on hardware; one example is where parallel and sequential processing are compared. The rest of the book concentrates on software—these vary over a whole range of matters such as complexity, strategy and stability in algebraic and other computational methods.

The book is well organised and physically well produced, presumably by a photographic method, and has a flexiback cover. It is rather specialised in its contents, but for those in the field it is a book that certainly deserves a place on the bookshelf.

F. H. George (Uxbridge)

Digital System Design Automation—Languages, Simulation and Data Base, by Melvin A. Breuer; 1977; 417 pages. (*Pitman*, £13·95)

This book is aimed primarily at programmers whose function is to provide software for computer hardware designers, and generally at anyone working near the hardware/software interface of digital systems.

Chapter 1, System Level Simulation, deals with conventional simulation languages. Implementation details are given for two such languages. The emphasis is mainly on the simulation of computer programs, and there is a large section on graph models of programs. Chapter 2, Register Transfer Languages and Their Translation, describes a register transfer language called DDL. In Chapter 3, Register Transfer Language Simulation, the problem of translating DDL descriptions into executable hardware simulation programs is tackled. A weakness of both these chapters is that too much emphasis is placed on minor details of a specific language; this tends to obscure more fundamental issues such as the problems of simulating parallel or asynchronous operations.

The fourth chapter, Design Automation Aids to Microprogramming, contains a good description of microprogramming, and illustrates the use of a register transfer language for describing the operating of a microprogram. The final chapter, Data Structures, Data Base and File Management, contains standard material on data base management and the representation of data.

The book has several weaknesses; in particular, it is excessively verbose. However there is a great shortage of literature on this subject, and the book fills an important gap.

Peter J. Moylan (Newcastle, Australia)