A

Manual

For

# SCALP

being a

Self Contained ALgol Processor

for the

General Precision

LGP-30

CCM-7A

Computation Center
Dartmouth College
Hanover, New Hampshire
1 January 1964

# Preface

This manual contains programming and operating information for the SCALP system, an squeezing of ALGOL into a load-and-go system for the LGP-30. Many restrictions obviously had to be made and these are listed in chapter 3. But a remarkable achievement is the number of features of ALGOL that are included. Such things as mixed expressions, nested conditionals, multipl assignments, and full subscript freedom are included. We have found that the only really troublesome restriction is not having the step-until type of for statement.

The planning, programming, coding, and debugging of the system were entirely the work of two former Dartmouth undergraduates--Anthony W. Knapp and Stephen J. Garland. A close study of the system and the coding sheets (which together with the block diagrams of the compiler are available) will almost certainly impress one with the superb and elegant job these two men did. Since SCALP is outgrowth of the earlier but not completed ALGOL-30, we should also acknowledge the contribution of two other former undergraduates--Robert F. Hargraves and Jorge Llacer--to that effort.

The coding sheets for the full system (but not the subroutine library) together with the block diagrams of the compiler are available as memoranda ccm-8A and ccm-8B, not respectively.

This manual does not pretend to be an ALGOL primer, treating that language only cursorily. The assumption is that ALGOL is either known or can be easily learned from any one of an increasing number of readable texts. For instance, see McCracken, A Guide to ALGOL Programming, Wiley, New York, 1962.

Thomas E. Kurtz
Hanover, New Hampshire
13 December 1963

Table of Contents

# I

# WHAT IS A PROGRAM

## 1.1 Introduction

Computers are still considered by many people to be ysterious machines capable of almost supernatural powers in solving vast and complex problems. Most of the references to computers in the various mass media have done nothing to counter this attitude. For instance, one is impressed with the almost human response of a computer printing its predictive messages on election nights.

In point of fact, a computer is nothing more than a device for very rapidly performing such elementary tasks as adding two numbers, dividing two numbers, or ascertaning whether a given number is negative or not.

To put it another way, a computer can do little more than a small child trained in only elementary logic and arithmetic, except that a computer is faster by 5 or 6 orders of magnitude. (An order of magnitude is a power of ten.)

This huge speed ratio of 100,000 or a million to one is the important factor that makes computers so useful. They have no innate intelligence, but very complex "intelligent-like" behavior can be simulated by performing millions of elementary logical operations in a short time, and this is where the computer's power lies. They have been dubbed by a well-known computer expert as high speed idiots.

Referring to the election night performances, all the computer is doing is making simple calculations of odds based on a selected number of early returns, and printing out messages that have been previously entered by human operators. The really difficult part of election forecasting--analysis of effects of local issues, development of statistical techniques for combining data, and rapid collection of results--are not done by the computer.

In fairness, it should be stated that computers do solve vast and complex problems, but these problems are of a nature that is both more important and less dramatic than the public realizes. Election forecasting is dramatic, but automatic control of production processes is a vastly more important application which has

immense and far-reaching implication.

1.2 What is a Program

Let us imagine that we wish to attack the simple problem of solving a quadratic equation. Except for occasional errors, each of us would arrive at essentially the same answers, though we might employ quite different methods.

The steps that we go through according to our method constitute a program. And that is what a program is--a set of steps or instructions for going from data to answers. The notion of a program is more obvious if we utilize the services of our grade school pupil to do the actual computation. Whereas we take for granted the individual steps when we do our own computing, we must spell them out in complete detail if our pupil is to arrive at answers. Furthermore, this spelling out must be done in a way that the pupil can understand.

As an example, suppose we supply our pupil with a worksheet containing fourteen columns and as many rows as we have quadratic equations to solve. Further, suppose that the coefficients a, b, and c ($ax^2 + bx + c = 0$) are listed in columns 1, 2, and 3, respectively. Then a program might go as follows:

"Perform the following steps in order:

1. Take this worksheet containing three columns of numbers and go to your calculator.
2. Take the first row and mark it with a little check.
3. For steps 4 through 17 look only at the numbers and spaces in the checked row.
4. Multiply column 1 by column 3 and write the answer in column 4.
5. Multiply column 4 by the number "4" and write the answer in column 5.
6. Multiply column 2 by column 2 and write the answer in column 6.
7. Subtract column 5 from column 6 and write the answer in column 7.
8. If the number in column 7 is negative, skip the next few instructions and proceed, starting with instruction 16; otherwise, proceed with instruction 9.
9. Take the square root of column 7 and write the answer in column 8.
10. Change the sign of column 2 and write the answer in column 9.
11. Subtract column 8 from column 9, and write the answer in

column 10.
12. Multiply column 1 by the number "2" and write the answer in column 11.
13. Divide column 10 by column 11 and write the answer in column 12.
14. Add column 8 to column 9 and write the answer in column 13.
15. Divide column 13 by column 11 and write the answer in column 14.
16. Erase the check mark and mark the next lower row.
17. If the check-marked row has no coefficients in it, proceed to step 18; otherwise, go back to step 3.
18. Bring the worksheet to me."

Of course, the real roots, if any, will be found in columns 12 and 14.

If we are more fortunate, we might call upon a more skilled person who knows about quadratic equations. We might then be able to supply him with the list of coefficients and verbally tell him to:

"Solve these quadratic equations." A bit later we will get back the desired answers, which, with a bit of luck, will be correct.

Both of the above sets of directions are programs. The resulting sets of answers will agree, up to round off, even though the "computers" were different. Notice that both programs were in English. Also notice that the first program was given in terms of tasks that either person could perform, and the second one in tasks that only the more advanced person could understand.

Thus, though programs for doing the same problem may differ widely depending on the type of computer, two general characteristics are common to all programs:

1. The program must be presented to the computer in a language he "speaks."
2. The program must be presented in terms of the elementary operations understood by the computer.

If the computer is electronic rather than human, the computation will be done more quickly but the program will have the same features as those above. It will be a set of directions given

in the language of the computer and in terms of operations the computer can perform.

1.3 A Program on an Electronic Computer

Present day computers operate at roughly the level of the first, longer program. That is, they can add, subtract, multiply, and divide, and they can test the sign of the number. Therefore, a program for solving a quadratic equation might start something like this:

```
  .
  .
  .
BRG  1
MPY  3
STØ  4
BRT  4
MPY  15
STØ  5
  .
  .
  .
```

This portion corresponds to steps 4 and 5 in the program given earlier, and assumes that column 15 contains the constant number 4. The program would vary considerably from computer to computer, but would remain similar in its essential characteristics.

How much nicer it would be to have a computer equipped with a microphone and a photo-reading machine. We might then simply place our coefficeints in the reader and speak into the microphone "Solve these quadratic equations." While such a system is not yet available, work on sound recognition and character recognition is in progress, and should be available in five or ten years. The striking fact, though, is that the computer used with these devices will be the same as those in use today. The trick used is to make a standard computer act like a voice-recognizing computer by supplying it with a special complicated program. This special program makes the computer act like a different computer capable of carrying out complex tasks.

Getting back to the example, the most reasonable type of human computer to expect is one who can understand standard algebra formulas but still doesn't know about quadratic equations. The program to the human computer might be:

"Perform the following steps in order.

1. Take these sets of numbers, which we might call a, b, and c, and go to your calculator.

2. Take the first of these sets of numbers.
3. Calculate $(-b-\sqrt{b^2-4ac})/(2a)$ and call the resulting number root1.
4. Calculate the same thing but with the second minus sign changed to a plus sign and call the resulting number root2.
5. Write down the value of root1 and root2 alongside the set of a, b, and c from which they were computed.
6. Take the next set of a, b, and c and repeat, starting from step 3 above.

When done, bring me the worksheet."

In the next section is presented a version of the program suitable for a certain computer that "understands" algebraic expressions but not quadratic equations. Of course, we are talking about a computer that has been "programmed" to act like a special computer that can "understand" these expressions. The language used is a very specific one call ALGOL. It was invented between 1958 and 1960 by committee of computer experts from several different countries. The idea of an "Algebraic" language for a computer is not new, going back to FORTRAN on the IBM-704 and MATH-MATIC on the Univac I. Since the inception of these languages in the middle 1950's, there has grown a plethora of similar languages, at least one different one for each major type of computer. It was this "babel" of languages that motivated the development of ALGOL as a standard language.

There are certain differences between ALGOL as it is defined and the language used with the SCALP system. These differences are imposed by the limitations of the equipment, and would be different for different computers. Therefore we first give an example in the ALGOL language, and then show how it would look as prepared for SCALP.

## II

## A Brief Look At ALGOL

The basic features of ALGOL are perhaps best presented through an example. The problem discussed in the Introduction might be incompletely phrased in ALGOL as

```
for i := 1 step 1 until n do
begin disc := b[i]↑2 - 4*a[i]*c[i]
     if disc < 0 then go to none else go to next;
     next: z := sqrt(disc)/(2*a[i]);
     root1[i] := -b[i]/(2*a[i]) - z;
     root2[i] := root1[i] + 2*z;
     none: end
```

Examining this program, we see that it is composed of statements each one ending with a semi-colon (rather than a period). Statements may be compounded, in which case the symbols "begin... end" are used to denote the start and the finish of the compound statement or "paragraph". (For clarity, we have indented the components of a compound statement, but such indentation is not a part of ALGOL.)

The underlined words are part of the ALGOL vocabulary, as are all the symbols such as ":" and "+". The non-underlined words refer to data or to labels in the program. For instance, "disc" refers to the number produced when the discriminant of the equation is computed; "next" serves to label a particular statement in the program and does not refer to any data.

Statements may be of several types. The one in the second line is an assignment statement because it assigns to the data location "disc" the value produced when the expression to the right of the ":=" is evaluated. There are several other examples of assignment statements in the program, and they all have the

characteristic that the variable appearing on the left of the ":=" symbol is assigned the value associated with the expression to the right of the":=". Most parts of the right-hand sides are self-explanatory: "*" stands for multiply; "-" for subtract; "/" for divide by; and "+" for add. The up-arrow stands for exponentiation or raising to a power. (The official ALGOL symbol for multiplication is a $\times$, but because it is similar to a capital X we have taken the liberty of using a * instead.)

Square brackets denote subscripts. In other words, b[i] is the ALGOL way for writing $b_i$, the $i^{th}$ value in a list of b's. The a's and c's similarly refer to lists of values with the subscript "pointing to" the particular value desired. We assume, of course, that these lists of values are in the "computer" where they can be used. root1[i] and root1[i] also refer to lists in which we will find our answers after the computation is completed.

As in algebra, parentheses are used when necessary to clarify an expression. In the line labelled "next" we wish to divide by the quantity $2a_i$, so we enclose 2*a[i] in parentheses. Another example:

a↑b+c stands for $a^b+c$ or $(a^b)+c$

while

a↑(b+c) stands for $a^{(b+c)}$ .

In the forth line we extract the square root of the quantity called "disc" by enclosing it in parentheses and preceding that by sqrt. Other standard functions are included in ALGOL-- a partial list follows:

| | |
|---|---|
| sqrt(E) | square root of the value of E |
| sin(E) | sine of the value of E |
| cos(E) | cosine of the value of E |
| ln(E) | natural logarithm of the value of E, |
| exp(E) | exponential, base e, of the value of E, |

where E stands for any expression that has a value.

The non-underlined words are called "identifiers"; they consist of any letter followed by a string of letters or digits or a mixture of the two. In the sample program, identifiers are used in two ways: the first as names of variables -- n, disc, a, b, c, z, root1 root2; the second as "lables" for statements -- "next" and "none" are used to label the statements that follow them. Notice that a statement label is separated from its statement by a ":". A string of digits with no letters may also serve as a statement label. We next see why we need statement labels in the example program.

The third line is a conditional statement whose meaning is almost self-explanatory; namely, if the discriminant of the equation is negative, then next execute (go to) the statement labelled "none"; otherwise next execute(go to) the statement labelled "next". The label following the go to serves to identify the statement we intend to be next obeyed. After all, if "disc" is negative, the equation has no real roots. The form of the conditional statements may vary widely according to needs and tastes, but the grammar of ALGOL insists that the words if, then, and else be used, and be used in much the same form as in the example. The statements constituting a program are normally executed in sequence; the sequence is broken only by go to statements.

Another type of statement is the "empty" statement that is used primarily to label an end. In the program example, each time disc is negative we go directly to the end of the compound statement that constitutes the body of the computation, and so we must provide a label. Since no further computation, is necessary, the statement so labelled is "empty".

This brings us to the for statement. An English translation of the first statement of the example is: "Do the statement immediately following the word do (it's a compound statement occupying lines two through seven), first with i set equal to 1, next with i set equal to 2, and so on, and finally with i set equal to n. After that execute the statement following."

The variable i is the subscript of the variables appearing in the compound statement. The computation is done first with $a_1$, $b_1$, $c_1$, $root1_1$, and $root2_1$, and then with $a_2$, $b_2$, $c_2$, $root1_2$, $root2_2$, and so on. The for statement is very powerful, allowing us to write in a very few steps computational programs that are otherwise very long. The specification of the for statement allows forms other than that one used in the example; for instance, an equivalent form is

"for i := 1,2,3,4 step 1 until n do ..."

or, if n is equal to 10,

"for i := 1,2,3,4,5,6,7,8,9,10 do ..."

Even other forms are possible, as a more complete study of ALGOL would show.

You will note that there are no statements in the example referring to imput or output. ALGOL itself does not concern itself with such statements because they will vary from computer to computer. In the example we have assumed that the variable n has been assigned a value equal to the number of quadratic equations to be solved, and that the lists a, b, and c have been given values equal to the coefficients of the equations. We also assume that the lists root1 and root2 are available to the user after the computation has been completed.

Notice that the variables i and n take on only integer values while disc, z, and the other variables can take on any real value. (In a computer the non-integers are represented as floating point numbers.) This fact is declared to the program by the declaration statement "integer i,n;". The reason for this declaration is that digital computers are frequently constructed to handle integers or whole numbers differently from general real numbers.

Because this general program may be used many times, we must also declare the miximum number of equations that we will ever want to solve. (This maximum number is not the same as n, which tells

the program how many equations there are this time. This declaration allows us to determine ahead of time whether or not the program can be fit into the computer memory; if not, then some modification will be made. In ALGOL we may declare the maximum size of the lists in the form "array a, b, c, root1, root2[1:1000]". The array declaration so-called, states that the lists (or arrays) a, b, c, root1, and root2 all have the same size, and each one provides a place corresponding to subscripts running from, for example, 1 to1000.

The user can supply comments and instructions for his own information or the information of others by using the symbol comment. Such comments do not affect the computation.

With the addition of comments, array declarations, and variable type declarations, the example program appears below. The program itself is set off by a begin-end pair and the various declarations, such as "integer i,n" appear immediately after the initial begin.

```
comment this algorithm finds the two real roots,
if they exist, of a series of quadratic equations;
begin integer i, n; array a, b, c, root1, root2[1:1000];
for i := 1 step 1 until n do
begin disc :=b[i]↑2 - 4*a[i]*c[i];

    if disc < 0 then go to none else go to next;
    next: z:=sqrt (disc)/(2*a[i]);
    root1 [i] := -b[i]/(2*a[i]) -z;
    root2[i]:=root1[i] + 2*z;
    none: end   end
```

For a more complete discussion of ALGOL the reader is referred to an excellent expository books by McCracken [A Guide to ALGOL Programming, Wiley, New York, 1962] or the technical definition of ALGOL [Peter Naur (ed.), "Revised Report on the Algorithmic Language ALGOL 60," ACM Communications, V.6(1963), n. 1, pp 1-17.] We will next consider how the above example would look in SCALP.

# III

## The SCALP Language

### 3.1 The Example

The SCALP language is basically a subset of ALGOL with certain modifications necessitated by the nature of the LGP-30. The language will be introduced first with an example. The example used will be the quadratic-solving algorithm given earlier with two important changes: input and output statements will be added, and the for statement will be eliminated so that the program will solve only one equation at a time.

```
comment'This algorithm finds the two real
roots, if they exist, of a quadratic
equation';'
begin'real'a',,'b',,'c',,'disc',,'root1',,'root2',,'z';'
start'::'read'('a',,'b',,'c')';'
disc':='b'Δ''2''-''4''*'a'*'c';'
if'disc'lt''0''then'goto'none'else'goto'next';'
next'::'z':='sqrt'('disc')'/'(''2''*'a')';'
root1':='-'b'/'(''2''*'a')'-'z';'
root2':='root1'+''2''*'z';'
print'('ccarr.',,'a',,'b',,'c',,'rcot1',,'root2')';'
goto'start';'
none'::'print'('ccarr.',,'a',,'b',,'c')';'
title'('c u.c.'n'c l.c.'o rea'l roo'ts.')';'
goto'start'end'
```

In addition to the inclusion of input-output statements and the deletion of the for statement, several minor changes will be noted. The most important is that the flexowriter stop code is used as a symbol separator. Others include representation of

characters not on the flexowriter keyboard, and a special way of designating numbers. These and others will be described in detail in the sections that follow.

## 3.2 How SCALP differs from ALGOL

In this section will be given the major ways in which ALGOL is restricted by SCALP. The examples illustrating the restriction will be given in ALGOL to avoid confusion with the stop codes and symbol representations in SCALP.

### 3.2.1 Blocks and Procedures

SCALP permits no blocks or procedures. The program itself, however, is a block and all variables used must be declared. Since there are no blocks or procedures, the concepts of recursive procedures, own arrays, and dynamic arrays do not arise. All arrays in SCALP must be declared using numerical integer constants to indicate the subscript bounds. The word own is not a part of SCALP, and may be used as an identifier, as may all other ALGOL words omitted from SCALP. (See section 3.3.1 and 3.3.3.)

While procedure declarations are not allowed, provision has been made for the standard functions, a list of which follows:

| | | |
|---|---|---|
| sin | sqrt | read |
| cos | abs | print |
| arctan | sign | title |
| ln | entier | |
| exp | random | |

The definitions of most of these functions are explained in the ALGOL-60 Report [ACM Communications, May 1960 and January 1963] and will not be included here. random is a no-argument function which when called generates a floating point pseudo random number in the interval [0, 1). To obtain a pseudo random digit from the set (0, 1, ..., 9), use entier (10*random). Read, print and title are input-output functions explained in sections 3.3.6, 3.3.7, and 3.3.8. The last three as well as the sqrt function appear in the example program.

### 3.2.2 Conditional Expressions

Conditional expressions are not permitted. However, conditional statements may be used in full generality. Thus

```
x:= y + if a < b then 3 else z;
```

is not allowed, and might be replaced by

```
if a < b then w:= 3 else w:= z; x:= y + w;
```

Along the same line, the arguments of a switch declaration must be simple labels. Thus, an argument of a switch declaration may not be another switch name or a conditional designational expression. If it is desired to employ nested swiches, the second switch must have a labelled call, the label appearing as an argument of the first switch declaration. Thus in place of

```
switch alpha:=goof, beta [n], stop;
switch beta:=loop, exit;
```

we might use something like

```
switch alpha := goof, L1, stop;
switch beta := loop, exit;
L1 : go to beta[n];
```

### 3.2.3 For Statement

Only a limited type of for statement is allowed in SCALP. Both the step-until and the general list are omitted. The only type permitted is the while element, which must be of the form

```
for <variable> := <expression> while <relation> do S;
```

where S is the iterated statement. Admittedly, the step-until element is probably the most useful. It may be imitated as follows: instead of

```
for i:= 1 step 1 until n do S;
```

use

```
i: =0;
for i:= i + 1 while i ≤ n do S;
```

Another form avoids the use of for altogether:

```
      i := 1;
loop: S;
      i := i + 1;
      if i ≤ n then goto loop;
```

This last form though a bit inconvenient to use, is actually executed in SCALP at about the same speed as the while form.

### 3.2.4 Special Integer Divide

The special integer divide* is not included in SCALP. It may be synthesized as follows: for

```
a * b
```

use

```
sign (a*b)*entier (abs (a/b))
```

### 3.2.5 Boolean

Boolean variables and operators are not allowed. One of the most common constructions which call for their use is

```
If x ≤ 2 ∧ x ≥ 1 then goto A else goto B;
```

to see if x lies in the closed interval [1,2]. Since the ∧ (or its transliterated form and) is not permitted in SCALP, one might use instead:

```
      if x ≤ 2 then goto L1 else goto B;
L1:   if x ≥ 1 then goto A else goto B;
```

## 3.3 Adaptation of SCALP to the LGP-30

The restrictions presented in the previous section are very severe, but affect the majority of programs surprisingly little. Many programs can be adopted to these restrictions. On the other hand, the use of stop codes and special symbol representations present no additional logical restriction, but are necessarily a part of all programs prepared for SCALP. They are itemized below, not necessarily in order of importance.

### 3.3.1 Symbols

Certain ALGOL symbols are absent from the flexowriter keyboard and must be represented by other symbols.

| ALGOL | SCALP |
|---|---|
| × | * |
| ↑ | △ |
| < | lt |
| ≤ | lte |
| = | == |
| ≥ | gte |
| > | gt |
| ≠ | =/ or /= |
| ' | '' |
| 10 | ten |
| : | :: |
| ↑ | ↑ |

In addition, the following symbols are excluded from SCALP:

÷ ≡ ⊃ ∨ ∧ ¬ ‘ ’

step until own label value

Boolean procedure string

Spaces have an effect in SCALP and should be avoided except in comments and title statements. go to may be used with or without the space.

### 3.3.2 Upper and Lower Case

Upper and lower case shifts should be used to make the typed copy more readable. They have no effect on the program to be run.

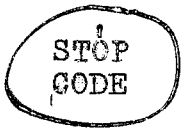
### 3.3.3 Restricted Words

No SCALP word may be used as an identifier (variable name or statement label). These include: go to, goto, if, then, else, for, do, while, comment, begin, end, integer, real, array, switch, sin, cos, arctan, sqrt, ln, exp, abs, sign, entier, random, read, print, title. Otherwise, full freedom for variable names is permitted except than only the last five characters are used. Thus squareroot and cuberoot will look the same to SCALP, since the last five letters are the same in each case. Furthermore, words

like witch that conflict in their last five letters with official SCALP words, in this case switch, may not be used as identifiers.

### 3.3.4 Symbol Separator

All SCALP symbols, labels, and variable names are separated by stop codes, the key on the flexowriter labelled




Dyads like := are considered single symbols, as are all SCALP words like begin, goto, and lte. Stop codes must not be omitted or misused, and great care in preparing the source tape must be exercised.

### 3.3.5 Number Representation in SCALP

Numbers may be included in SCALP programs. Any legal ALGOL number is permitted except that the following conventions must be followed:

1. The number must be started and finished with an extra stop code, so that a double stop code separates numbers from adjacent symbols.

2. There must be a stop code at least every five characters in the number, but complete freedom as to how much more often they might be used is allowed.

3. The letter string ten is used in place of the ALGOL subscript $_{10}$.

4. The significant figure part must not exceed eight digits, excluding sign and decimal point. (Since only about 7.5 significant digits are available for computation, this restriction causes no loss of accuracy.

5. The digit 1 is represented by an l, a lower case L.

The following examples are correct in SCALP:

```
x':='a+''2';'
mat'['-1'',,'i'+''+3'']'
c'*''14.29''+'d'
xyz'/''+.357'e'en-'5''
```

3.3.6 Input

Numbers and data may be entered when the problem is being run by using the read function. The following format is used in your program:

```
read'('var1',,'var2',,'varn')';'
```

There may be any number of arguments, and each may be a variable name representing a simple or subscripted variable of type either real or integer. For example,

```
read'('x',,'i',,'matric'['i'+''1'']',,'bla')';'
read'('n')';'
```

At run time the read function will cause the system to call for data input through whichever input device is connected (by means of the toggle on the photoreader). Small amounts of data may be entered via the flexowriter, but large amounts should be entered through the photoreader for efficiency.

On the data tape numbers are typed exactly as outlined in section3.3.5 except the leading stop codes are not needed. Thus, a data tape for reading the numbers

2, -10256, 2.79, -23, 3.14159

would be

```
2''-10'256''2.79'ten'-23''3.141'59''
```

Notice that the double stop code is used to finish a number, just as with numbers in your SCALP program, but there are no stop codes at the beginning of each number.

The number on the data tape need not agree in type with the corresponding argument of the read function. Thus, if the program is

```
begin'integer'n';'
read'('n')';'
```

and the number -13.3''is entered, n will be assigned the rounded integer value -13.

### 3.3.7 Output

Numbers are printed in SCALP by means of the print function. As with read, there may be any number of arguments and each one may be any type. Arithmetic expressions may also appear as arguments of the print function (which is not true of the read function). The following is a correct use of the print function:

print'(x',,'n'+''2'',,''17'')';'

Expressions of type real are printed in floating point in the form

±.xxxxxxx ±yy ,

where ±.xxxxxxx stands for the seven digit fraction and ±yy stands for the power of ten. Integer expressions are printed in the form

±nnnnn

where nnnnn is an integer that may have leading spaces. In all cases, the plus signs are omitted, so that the constant pi would come out

.3141592 01

and -17 would come out as

- 17 .

After each number is printed, SCALP executes a tab on the flexowriter.

### 3.3.8 Titles

Alphabetic information for labelling output data and for other purposes may be printed at run time by the title function. This function treats its argument as a letter string which it prints out almost exactly as given. There are two exceptions:

1. There must be stop codes at least every five characters, except as indicated in exception 2.

2. Typewriter controls are indicated by special code words six characters long which must be preceded and followed by stop codes. These codes are:

| | |
|---|---|
| carriage return | 'ccarr.' |
| upper case | 'c u.c.' |
| lower case | 'c l.c.' |
| tab | 'c tab.' |
| backspace | 'c b.s.' |
| color shift | 'c c.s.' |
| stop code | 'c stop' |

(The single space appearing in the last six codes is absolutely essential.) The entire string is enclosed in parentheses. The following shows a correct use of the title function:

```
title'('c u.c.'t'c l.c.'his i's a g'ood p'rogra'm.')';'
```

The printed result will be

This is a good program.

The two typewriter controls 'ccarr.' and 'c tab.' may also be inserted as arguments in read and print statements. They must, of course, be set off by commas, just as if they were variable arguments. The following examples are correct:

```
read'('ccarr.',,'x')';'
print' ('ccarr.',,'x',,'y',,'c tab.',,'x')';'
```

Attempting to insert any of the other typewriter controls into read or print statements will cause an illegal variable error stop.

### 3.3.9 Comments

The comment conventions of ALGOL are retained in full in SCALP.

## 3.4 Another Example

To show more clearly how one adapts an ALGOL program to SCALP, we look at Algorithm 112 [ACM Communications, V.5(1962), August, p 434.] The original algorithm as corrected, stripped of its explanatory comment and procedure heading, and with the formal parameters included in ordinary declarations, follows:

```
begin integer i, n; real x0, y0; array x, y[1:100];
Boolean b, answer;
x[n+1] := x[1]; y[n+1] := y[1]; b := true;
for i:= 1 step 1 until n do
     if (y0 < y [i] ≡ y0 > y[i+1]) ∧
          x0-x[i]-(y0-y[i])×(x[i+1]-x[i])/(y[i+1]-y[i])< 0
     then b := ¬b;
answer:= ¬b end
```

Notice that this algorithm contains (1) Boolean variables, (2) Boolean operators, and (3) a step-until element. We must eliminate all these features to be able to use SCALP, which we do as follows:

(1) Declare b and answer to be type integer and let +1 be true and -1 be false.

(2) Rewrite the long conditional statement into a series of conditional statement to eliminate the Boolean operators.

(3) Replace the step-until element by a while element.

All these changes are made in the modified algorithm below:

```
begin integer i, n, b, answer; real x0, y0; array
      x,y [1:100];
x[n+1] := x[1]; y[n+1] := y[1]; b:= 1; i := 0;
for i := i + 1 while i ≤ n do
begin if y0 < y[i] then goto L1;
      if y0 > y[i+1] then goto no;
L2:if x0-x[i]-(y0-y[i])×(x[i+1]-x[i])/(y[i+1]-y[i]) <0
   then goto okay else goto no;
L1: if y0> y[i+1] then goto L2 else goto no;
okay: b := -b;
no: end;
answer := -b end
```

Finally, we replace × by *, < by lt, ≤ by lte, > by gt, : by ::, ↑ by 1, comma by double-comma, and insert stop codes to get a legal SCALP program.

```
begin'integer'i',,'n',,'b',,'answer';'real'x0',,'y0';'
     array'x',,'y'['1'::'100']';'
x'['n'+'1']':='x'['1']';'
y'[n'+'1']':='y'['1']';'
b':='1';'
i':='0';'
for'i':='i'+'1''while'i'lte'n'do'
begin'if'y0'lt'y'['i']'then'go to'L1';'
     if'y0'gt'y'['i'+'1']'then'go to'no';'
     L2'::'if'x0'-x'['i']'-'('y0'-'y'['i']')'
               *'('x'['i'+'1']'-'x'['i']')'
               /'('y'['i'+'1']'-'y'['i']')'lt'0'
          then'go to'okay'else'go to'no';'
     L1'::'if'y0'gt'y'['i'+'1']'
          then'goto'L2'else'go to'no';'
     okay'::'b':='-'b';'
     no'::'end';'
answer':='-'b'end'
```

The above example, though perhaps a bit hard to plow through, illustrates how to adjust ALGOL programs for features of ALGOL not included in SCALP. This example is a "bad" one since it includes most features that SCALP excludes. You will find that for most common problems the inconvenience will not be nearly as great.

# IV

## How to Run a Program in SCALP

### 4.1 Introduction

We will assume that the reader is able to use a flexowriter off-line to prepare his source program tape. This preparation is straightforward, but is is well to remember the input characteristics of the LGP-30. Basically, all letters, digits, and punctuation can be inputted, but typewriter controls cannot (except for TAB, which enters as a binary 000000 when in 6-bit mode.) Thus, we have these rules:

1. There is no distinction between upper and lower case.

2. Spaces count as characters, so avoid their use except in titles. Also, you may use "go to" in place of "goto".

3. Do not use backspace to over-type certain characters, since the resulting hard copy is very difficult to proofread. (Originally, it had been intended to use the backspace to have the double comma ",," look like a single comma "," , a "::" like a ":", a "=/" or "/=" like a "≠" and an "==" like an "=". However, the slight imporvement in readability seems not worth the extra effort.)

4. You may use tabs to indent portions of your program, but be sure the tab does not follow any non-blank character except a stop code.

You will also want to have your data tape prepared ahead of time. (For very small amounts of data, you may use the flexowriter keyboard. See section 4.5.)

### 4.2 Loading the SCALP System

If the SCALP system is in the memory, then you may ignore this section. Sometimes it is hard to tell whether it is or not, but the best advice is this: if you are having trouble that you cannot

pinpoint, it may be that the system has been "damaged," and you should not hesitate to reload it. It goes without saying that a photoreader is assumed, in which case the system may be loaded in about four minutes without checksums or six minutes with checksums. If you don't have a photoreader, you will need a special loader. (The loader that comes with the system uses i0000 orders only to input, and this works only with the photoreader.)

The SCALP system tape is loaded by its own loader, which is brought in with a i0000, c0003, i0000 short bootstrap. The loader is a one-track hex loader that goes into track 58. This loading is done in 4-bit, in the normal manner. If the TRANSFER CONTROL switch is up (off), the loading is done with check sums at the end of every track. If this switch is depressed (on), the check summing is ignored and the reading speed increased. After loading, the loader may be overwritten by matrix storage.

There are several ways to install the short bootstrap at the bottom of memory. They are give below:

1. If the SCALP system is in momory and relatively intact, you may obtain the short bootstrap automatically.

   a. 6-bit down, typewriter on, OCNS.
   b. Type "bootstrap".
   c. Press START
   d. Load system tape upside down in photoreader, set input toggle to READER.
   e. Release 6-bit switch.
   f. Press START

2. If P.I.R. 10.4 (or similar version) is intact in memory, you may obtain the bootstrap by inputting this sequence:

```
;0000000',0000003'i0000'c000j'i0000'
```

Placing the system tape upside down in the photoreader, turning to READER, and executing OCNS will start the reading.

3. If there is nothing but junk in the memory, you use the manual input mode as follows:

a. Typewriter on.
b. MANUAL, "c0000", FILL, "i0000", ONE OP, EXECUTE.
c. MANUAL, "c0004", FILL, "c0003", ONE OP, EXECUTE.
d. MANUAL, "c0008", FILL, "i0000", ONE OP, EXECUTE.

(Locations 0, 1, and 2 should now contain i0000, c0003, i0000, respectively.)

e. Place system tape upside down in reader, turn to READER, and execute OCNS.

About the only comment to be made about this bootstrap is that it is similar to the old 10.4 bootstrap except (1) it goes into low memory where an OCNS can be used to reach it, and (2) the p0000's are missing.

## 4.3 Compiling

This description of compiling assumes the photoreader will be used. If you prefer to use the mechanical reader, you may do so without any modification to the compiler whatsoever.

Make sure the system is in memory, the typewriter is turned on and ready to go, and the photoreader is on and ready.

a. Load the source program tape upside down in the photoreader.
b. Set the input toggle to TYPEWRITER.
c. Depress 6-bit.
d. Perform OCNS.
e. Type "compile" and press START. (You are now at a breakpoint stop.)
f. Set the input toggle to READER, press START.

After about 8 seconds for resetting symbol tables, compilation begins. If an error stop is printed, you will probably have to locate the error on the source tape, correct it, and recompile from the start. Error stops are listed in a later section.

If there are no errors, the typewriter will type either "done" or "load". If it types done, you are ready to proceed with the running of the problem, and should turn to section 4.4. If it types "load", your program has used one or more of the

subroutines on the LIBRARY tape. These include the three power operations, and the special functions ln, exp, sin-cos, arctan, sqrt, entier, and random. (The functions sign and abs, and the input-output functions are included within the main system.) The library tape is loaded as follows:

g. Place the LIBRARY tape upside down in the photo-reader
h. Set input toggle to READER (if not already there.)
i. Release 6-bit, press START
j. After "done" is typed, depress 6-bit and go on to the running of your problem.

At step i the system gives you one extra chance to release the 6-bit button. If you had forgotton, it types "6-bit", whereupon you should perform step i again. If you forget after all this, you've had it! (Seriously, you may have to recompile your program.)

The entire library tape is read, but only those subroutines called for are actually placed in memory. Those not needed are speeded through the reader by a series of dummy i0000 orders.

The library tape must be loaded through the photoreader. Unlike compiling, which can be done through the flexowriter, loading of the library tape through the flexowriter requires a modification to the basic system tape.

If something happens to interrupt the reading of the library tape, you may recoup without the necessity of re-compiling by manually transferring to location 0037 (in hex, 0094). This is tricky, and if you aren't sure of yourself, it may be easier to recompile from the start.

One last remark. After any error message or other message printed by the system, the breakpoint stop is followed by a transfer to location 0000. Thus, pressing START will give you location 0000 more quickly than will OCNS.

## 4.4 Running a Problem

After a program has been compiled and the library tape loaded if necessary, the program is ready to run. If you have a data tape, it should be loaded in the photoreader. To get underway, do this:

a. Turn input toggle to TYPEWRITER.
b. Depress 6-bit (if not already).
c. Perform OCNS (or press START if a message has just been typed.)
d. Type "start", press START.
e. Depress BREAKPOINT 8 for continuous running.
f. Turn toggle to READER if your data tape is there.
g. Press START, and you are off and running.

All of the data input may be through the flexowriter reader or keyboard without any modification to the system. But rather than give detailed instructions for those who prefer to use the mechanical reader, we will assume that they understand the two input devices well enough to make the modifications to the above instructions by themselves. The same comment applies to compiling through the mechanical reader.

It should be emphasized that all data input is 6-bit, so that the only time the 6-bit button should be lifted is when loading the system tape or when loading the library tape.

## 4.5 Keyboard Programming

If you wish you may compose your program at the keyboard and not prepare a source tape. The same instructions are followed except that (1) the input toggle is kept at TYPEWRITER, and (2) wherever there should be a stop-code you must press START. Be sure that the MANUAL INPUT button on the typewriter is depressed. Also, be careful not to type anything or press START unless the input light is on. Be especially careful on data input--wait for the input light to come on before pressing the second START that signals the end of a number.

V

# Error Stops

Error stops will be divided into three categories -- those occurring at compile time, those occurring at run time, and others.

In all cases, a meaningful word or two is printed out, thus giving the user an indication of the trouble even before he consults these pages. The error stops are listed as they might occur, together with an interpretation and suggestions for correcting the trouble. In what follows, xxxxx stands for the offending identifier; that is, "error var. x" may mean that variable x has not been declared.

## 5.1 Compiling Errors

These errors can occur during compiling. In all cases, you will have to correct your program tape and recompile. There is no provision in SCALP for restarting-in-the-middle of a compilation. (For the curious, this is because the double push-down stack method of compiling doesn't recognize a statement as such. Consequently, when an error occurs, there may be extra or missing entries in the stacks with no way to backtrack to a previous semi-colon, for instance.)

| error stop | meaning and remedy |
|---|---|
| ovflo const | More than 30 different numerical constants used, not counting those in array declarations. Treat some constants as input data, or combine them. ("1" and "1.0" are different constants.) |
| ovflo expr | Symbol stack has overflowed because of too complicated an expression. Look for deep nesting of parentheses. Actually, it is the depth (of nesting) rather than the length that counts. Thus, x:=a + a + ... + a; (100 times) will compile easily, but x:= (( ... (a) ... )) ; (100 times will not.) |

| error stops | meaning and remedy |
|---|---|
| ovflo store | Storage for program and array data exceeded. Cut down on length of program, number or size of arrays, or number of size of switches. Since only 744 locations are available for program and arrays, you may have to segment your program into several subprograms, or go to a different computer. |
| ovflo symb | Total number of different labels and variable names exceeds capacity of symbol table (from 29 to 53, depending on random way symbol table fills out.) Eliminate unneeded labels, plan multiple use for working variables, use constants directly, or(as a last resort only) replace several variables by a single subscripted variable, |
| ovflo temp | Temporary storage exceeded. Replace long assignment statement by two shorter ones. |
| array xxxxx | Array xxxxx appears as an unsubscriped variable. Make sure all arrays are subscripted, check for a stop code between the array identifier and the left bracket, viz., xxxxx'[' . |
| array bound | Upper bound < lower bound. Check array bounds. Make sure all array bounds are given as integer constants. |
| declr xxxxx | Illegal symbol xxxxx in a declaration, or the symbol following a switch identifier is not a := . Check declarations. |
| label xxxxx | Symbol following a go to is not a label, or a label is defined twice. Check for using a variable or ALGOL word following a go to, or a multiply-defined label. |

| error stop | meaning and remedy |
|---|---|
| var.xxxxx | Symbol being used as a variable is not a variable. Check for undeclared variables, missing variables, or possibly labels being used as variables. |
| var. adj. | Variable or constant adjacent to a variable or constant. Look for missing operation symbols, especially* signs. May indicate lack of double stop code '' following a numerical constant earlier in the program, or misuse of comment convention. |
| var. ????? | A variable is missing somewhere in the source program. Look for adjacent operators like +*'. Actual error may be far removed from symbol causing the error stop. |
| xxxxx xxxxx | A syntactical error of some sort recognized in the double stack compilation as an impossible combination. The second symbol is the one just read in. Rembember that all upper case symbols will print in lower case; thus, 0 insted of ). Check use of parentheses, use of begins and ends, missing then, incorrect subscript formation, and incorrect for statement format (only the while type is allowed.) |
| digit | The significant figure part of a number has more than 8 digits, or the exponent has more than 2 digits. Round off to 8 or fewer figures, or scale so as to avoid extreme exponents. |
| input | Overflow on input; real number too large to represent in floating point. Number should be smaller than $2^{128} \approx 10^{38}$ in absolute size. |
| type | Variable preceding a [ has not been declared as an array. Check to see if all arrays have been declared. |

| error stop | meaning and remedy |
|---|---|
| title | The symbol following title' is not a (' . Check title statement. |

It should be noted that there are some not unlikely error conditions that do not have error stops and can lead to trouble. The know ones are listed here:

1. In an array declaration, the constants must be integers. If they ars real, or if you inadvertantly include a decimal point, the array bounds will be ridiculous and no warning will be flashed. This error could be detected at the expense of some valuable memory.

2. Undeclared left-part variables are not detected. Thus, if you use an undeclared variable, or a label, on the left side of the assignment symbol := , there will be no warning. Again, this could b e detected at a slight cost in memory.

The above two were the ones know at the time of this writing--there may be others.

## 5.2 Run Time Errors

Even if a program is grammatically correct, it need not produce the answers desired. The ways in which a program may be logically incorrect are legion, but there are a few conditions that can be detected and that may indicate trouble. For instance, trying to take the square root of a negative number or coming up with a number larger than $10^{38}$ suggest errors in programming. In all cases, the remedy is to reprogram, or to use the execute or patch features (see chapter 6) and continue.

| error stop | meaning and remedy |
|---|---|
| div 0 | Division by zero has been attempted in a divide operation or a power operation of the form r'Δ'i' where r = 0.0 is real and i ≤ 0 is an integer. |
| ovflo | A real (floating point) number is larger than $2^{128}$ in absolute value. This can occur only on a temporary or assigned hold. Thus, x':='a'+'b'-'c';'... with a=b=c=$2^{128}$ -1 will not cause an overflow. An underflow (a real number < $10^{-38}$ in absolute size) will not cause an error stop, but will replace the offending number by zero. |
| swtch | The argument of a switch call is not in range. If there are n positions in the switch declaration, then the argument must range from 1 to n, inclusive. |
| power | Error in a power expression. Look for 0Δ 0 where zeros may be of any type, 0Δi where is a negative integer and the zero is a type integer, or aΔb where a < 0, or a = 0 and b < 0 for real a and b. |
| ptype | Occurs in iΔj if i and j are of type integer, but j is negative and the next term is not of type real. Reprogram, arranging for the iΔj to be next to a real quantity. Actually, this error should not be possible since ALGOL permits full mixed expressions. However, in SCALP, the mode of the computation is determined at compile time while the type of iΔj is not known until run time. Thus, if no float instruction has been compiled following the Δ |

| error stop | meaning and remedy |
|---|---|
| | instruction, an error stop may occur. Below are examples where this can happen, where m and n are integer and x is real:<br><br>m' :=' n' Δ' n' ;'<br>x' :=' m' +' m' Δ' n' .'<br>x' :=' m' Δ' n' +' m' +' x' ;'<br><br>The error stop cannot occur in any of the following cases:<br><br>x' :=' m' Δ' n' ;'<br>m' :=' x' +' m' Δ' n' ;'<br>x' :=' m' Δ' n' +' x' +' m' ;'<br>m' :=' m' Δ' x' +' 0.0' ;' |
| print | Integer quantity>99999 in absolute size too large to print. Scaling integer arithmetic is difficult in SCALP. In particular, it is easily possible to generate an integer number that overflows the LGP-30 accumulator without a warning being flashed. (For those with overflow logic boards, there is no warning in any case; for whose with a standard machine, an overflow on add or subtract will cause an overflow stop at location 0246.) |
| input | See corresponding error stop in section 5.1. |
| digit | See corresponding error stop in section 5.1. |
| ovflo intgr | An integer ≥ 32,768 has resulted from an i Δ j operation, or is trying to become floated. |
| ovflo exp | The argument of the exponent function is larger than $\ln_e 2^{128} \approx 88.72$. |
| sin<br>cos | Number too large to allow computing the sine, or cosine, to even one significant figure. |

| error stop | meaning and remedy |
|---|---|
| sqrt | Argument of the square root function is negative. |
| entr | Number too large in absolute value ($\geq 32{,}768$) as argument of the entier function. |
| ln | Argument of the ln function $\leq 0$. |

About the only additional comment is that integer quantities can become meaningless without a warning having been given. For instance, work with factorials is severely limited in SCALP. If you have a problem requiring integer arithmentic with large numbers of figures, you may have to program in machine language on a variable word length computer.

## 5.3 Other Error Stops

| error stop | meaning and remedy |
|---|---|
| order | An illegal command to the system has been typed, or 6-bit is not down. (Commands to the system mean words typed after performing OCNS.) Check 6-bit. The only legal commands are "compile", "start", "bootstrap", "continue", "execute", and patch". The first three have been discussed; the last three will be discussed in chapter 6, Debugging. |
| 4-bit | Forgot to lift the 6-bit button before loading the library tape. Lift 6-bit and press START. |

# VI

# Debugging

## 6.1 Introduction

An important, and often the most time-consuming, part of any programming task is debugging. This is the process of detecting and removing logical errors in the program. (Grammatical errors are removed at compile time.) Even if no run-time errors are detected by the computer, the answers produced need not be correct. In general it is very difficult to say for sure whether or not the answers are correct, but often we can detect bad answers by their inconsistency with each other or with previous calculations. The causes of the errors (programming errors) are sometimes easily detected, often not so easily, or may be caused by round-off errors. In the last case, fundamental and theoretical questions are involved, and a complete rethinking of the problem may be needed.

Debugging has two phases -- finding the error (the hardest part), and eliminating it. SCALP includes the features now discussed to help locate the error and to change the program without reprogramming or recompiling.

## 6.2 Tracing

Normally, only the print and title functions can cause a printout. Tracing offers a way to inspect the intermediate calculations as well. Tracing in SCALP is carried on as long as the TRANSFER CONTROL switch is on (depressed.) Printed under these conditions are:

1. All statement labels as they occur.
2. The name of each variable appearing on the left side of an assignment statement together with its numerical value.

As an example, consider the following program segment where x is real and n is integer:

```
:
n':='1'';'x':='1.0'';'
loop':'x':='x'*'n';'
```

```
n':='n'+''1'';'
if'n'lte''3''then'goto'loop';'
next'::'
```

⋮

If TRANSFER CONTROL is down while this segment is run, the following trace print out will result:

⋮

```
          n          1
          x     .1000000      01
loop      x     .1000000      01
          n          2
loop      x     .2000000      01
          n          3
loop      x     .6000000      01
          n          4
next
```

⋮

It should be noted that only the variable name of a subscripted variable is printed; the subscripts themselves are not traced and must be inferred.

Though tracing can be time consuming on a long problem, it is the simplest and often the most effective debugging tool.

## 6.3 Stop and Continue

At any point during the running of your program you may stop it by lifting BREAKPOINT 8. Performing OCNS will then permit you to use the execute and patch features given in sections 6.4 and 6.5. Or, if you change your mind before performing OCNS, you can to on with your program by depressing BREAKPOINT 8 and pressing START. Or you can step through your program one instruction at a time by leaving BREAKPOINT 8 off (up) and pressing START for each instruction.

If you wish to continue your program, whether or not you have used the execute or patch features, do this:

a. Perform OCNS.

b. Type "continue".

c. Press START.

d. Depress BREAKPOINT 8 for continuous running.

e. Press START.

The program then continues from where it left off.

## 6.4 Execute

At any point during the running of your program you may stop its running and execute any SCALP statement you wish. You may execute as many statements as you like before continuing with your problem as described in the previous section. The procedure is this:

a. Lift BREAKPOINT 8.

b. Perform OCNS.

c. Type "execute", press START twice.

d. When the light comes on, type any legal SCALP statement. Be careful to press START for every stop-code encountered ( the stop-codes need not be typed at all, but it may be easier to do so than to change a habit.)

e. When you type a ;' indicating the end of the statement, it will be compiled, executed, and forgotten. After the breakpoint stop, pressing START will give you location 0000 (the same as thing that OCNS achieves) and you can then give another command (such as "continue" or "execute") to the system.

Some ways in which the execute feature may prove useful are:

1. Dumping: After the light goes on, typing print'(x')';' will cause the value of x to be printed. (Remember on the keyboard press START in addition to or in place of each stop-code.) More complicated expressions, including subscripted variables, may be printed.

2. Changing values: After the light goes on, typing x'!=:'' 7.53'429'';' will change the value of x to 17.53429. x'!=:''2''*'x';' which double the current value x. (Again, press START for each stop-code encountered.)

3. Jumping: You may transfer to any labelled statement in your program. For instance, begin'x'!=:''35.2'';'goto'loop'end';' will change the value of x to 35.2 and then transfer control to the statement labelled loop. Notice that since we wanted to execute two simple statements, we grouped them with a begin-end pair.

Some further comments about using execute:

1. Don't forget the semi-colon.

2. If the statement to be executed calls for a function or functions from the library tape, it must be loaded. The system does not remember that certain of these subroutines may already be in memory. Furthermore, the new copy of a duplicated subroutine will, in effect, replace the old copy. Since a second execute will wipe out the program left by the first, a crucial subroutine may be lost. To avoid this unfortunate situation, either (a) do not use constructions that call for the library tape, or (b) make sure the last executed statement utilizes all subroutines used in both the main program and in at least one of the previous executed statements. (This is an unfortunate trap that could be remedied with another track or two of memory to be taken from program and data.)

3. The original symbol table is kept intact, so do not use as labels any identifier used as a label in the main program. You may declare new variables, but do not make any declarations in conflict with the original declarations. If you declare new variables, you must also enclose the statement in a begin--- end pair.

4. It is entirely possible for an executed statement to require more memory than is left over. However, after each execution is completed, the available memory reverts to its

state at the end of compilation, except that the symbol tables and matrix storage assignments retain their new status. There is also the problem of the subroutines mentioned in item 2 above.

5. For most purposes, the possible problems mentioned in items 2 and 4 occur only rarely, and execute provides a simple way of performing many debugging tasks in the source program language.

6. It is possible to use execute to imitate the operation of a desk calculator, but such operation is not recommended.

## 6.5 Patch

Your program may be permanently modified by patching. Patching will actually insert the inputted piece of program. Physically, it goes at the end of your existing program, but is tied to it with transfers. You specify where the patch is to go by giving a statement label; if you have no labels in your program, you cannot patch. A sequence of patches adds permanently to the program in memory, and may exhaust memory.

A patch is a compound statement that is inserted between a statement label and the statement thus labelled. It must begin with a begin and end with an end, whether a single statement or not, and does not need a final semi-colon as does execute. New variables may be declared, but they should not conflict with previous declarations nor should defined lables be redefined.

The procedure for patching:

a. Lift BREAKPOINT 8.

b. Perform OCNS.

c. Type "pathc", press START twice.

d. Type the desired label, press START

e. Type the patch, starting with a begin

F. At the end of the patch, after typing end, control returns to location 0000 after a breakpoint stop.

As an example, suppose in a shile type for statement, you forgot to initialize the running variable:

```
          :
    loop'::'for'i'+''1''while'i'lte'n'do'S';'
```

where S is the iterated statement. You may correct by typing, starting with step d. above,

```
    loop'begin'i':='0''end'
```

The net logical result will be

```
          :
    loop'::'goto'patch1';'loop1'::'for'i':='i'+''1''...
          :
    patch1'::'i':='0'';'goto'loop1';'
```

Thus, insertions can be easily made, but they must always be tied to labelled statements.

Correcting an erroneous statement can also be done, but not as simply as can insertions. If we think of the program between two labels as a program segment, we must replace the entire segment containing the erroneous statement or statements with a patch that includes the corrected segment. As an example, suppose a quadratic solving routine appears in part thus:

```
          :
    xyz'::'root1':='-b'/''2''*'a'-'c';'
    root2':='root1'+''2''*'e';'
    here'::'print'('....
```

We quickly recognize that we should be dividing by 2a, not dividing by 2 and multiplying by a as shown. That is, we have forgotten the parentheses. To correct the error without recompiling, we call for patch. After typing "patch" and pressing START twice, we type

```
    xyz'begin'root1':='b'/'('2''*'a')'-'c';'
    root2':='root1'+''2''*'e';'
    go to'here'end'
```

Notice that the last statement in the patch is a goto.

Obviously, patching would be easier if our program were liberally sprinkled with labels.

If the label that identifies the patch is illegal, an error patch will be printed. Perform OCNS and start again.

Two points should be emphasized:

1. Do not type a ':' after the identifying label.

2. Patch requires a begin --- end but no semi-colon, whereas execute requires a simi-colon even if it has a begin---end.

# VII

## For Experts Only

This short chapter contains some facts that are not necessary to the running of the SCALP systembut that may be interesting to some readers. The title of the chapter permits us giving only the briefest of explanations for these facts.

### 7.1 With Reference to ALGOL

Even though chapter 3 illustrates only very simple ALGOL constructions, SCALP has all of ALGOL except those features listed in chapter 2, and certain minor restrictions mentioned in chapters 4 and 5 (integer constants must be used in array declarations, and the problem connected with error ptype.) Even a ↑ b, where a and b are integers and b<0, is performed by 1/(a.a...a) (-b times), so that a can be <0 as well. Mixed real and integer expressions, nested conditionals, and complete subscript freedom are allowed. Labels can be numeric or otherwise. Full freedom with the comment convention is allowed (except be careful to not set off an end, else, or; by stop-codes unless you mean to!) Nested arrays are allowed, such as a'['a'['i',,'j']',,'k']'. Iterated assignment statements are permitted to any depth.

### 7.2 With Reference to FORTRAN

Some users may be familiar with FORTRAN rather than ALGOL. For them, these comments apply:

1. Declare all variables in SCALP; in FORTRAN those beginning with I, J, K, L,M, or N are assumed integer.

2. Mixed expressions are not allowed in FORTRAN, but are in SCALP and ALGOL. For instance, AVE=SUM/N is not allowed in FORTRAN, but average':='sum'/'n';' is okay in SCALP.

3. SCALP and ALGOL use the semi-colon ag a statement separator. Thus, there is no need for the FORTRAN continuation convention.

4. Alphabetic labels are permitted (and encouraged) in SCALP and ALGOL.

5. ALGOL permits a much more general conditional or if statement. Despite the simplicity and usefulness of the FORTRAN if statement, occasional programmers sometimes find it hard to remember where to put the commas, and whether it is <,=,> or >,=,<.

6. ALGOL has a different form for the for statement, more geneal than the FORTRAN DO statement. (SCALP permits only a limited type of for statement, which is probably less convenient than the FORTRAN DO.)

7. Arrays and subscripts are limited in FORTRAN to three-dimensions and integer expressions of the form c1*v ± c2, where the c's are constants and the v is an integer variable. No such restrictions in ALGOL-SCALP.

## 7.3 Compiling

Compiling is done with a double stack last-in, first-out algorithm adapted from Samelson and Bauer. The symbol table follows the suggestion of E. J. Williams. The details are too numerous to include here, although the coding sheets and block diagrams might give some information.

The compiler writes the object program directly into memory, and entirely in an interpretive language. Even integer operations are interpreted. (This means that the simplest subscripted variable requires about 0.5 second per subscript position at run time.) Divide-into and subtract-from instructions are used, as well as float, round, and a series of arithmetic operations that float an integer quantity as it comes form memory. There are two kinds of holds for each type of variable, one that can be traced (used for assignments), and one that can't be traced (used for temporary holds.)

This remark should go into chapter 6: If you interrupt an execute to start another execute, you will end up in an execute loop when you type "continue".

## 7.4 Running

The interpreter recognizes 52 different instructions. Besides the integer and floating point instructions, each special function has a corresponding instruction. All integer arithmetic is interpretive, but no precautions for overflows are taken. All floating point is unnormalized except before holds and when entering subroutines; this speeds things up with no loss in accuracy in virtually all cases.

No extensive timing calculations have been run, but some estimates have been made. These must be considered as accurate to about 1.5 significant figures:

| | | |
|---|---|---|
| floating add | .390 | seconds |
| floating multiply | .230 | " |
| subscripts, each(min.) | .540 | " |
| integer add | .120 | " |
| integer miltiply | .190 | " |

No estimates have yet been made of the subroutine running times for the special functions.

## 7.5 With Reference to ACT III

Although it was not the intention to have this ALGOL system compete speed-wise with existing systems, the urge to compare it with ACT III is irresistable.

For ease of use, SCALP should be compared with ACT III, mode 2A, a stripped-down load-and-go version. SCALP can handle larger programs in some cases than can ACT III-2A, and is easier to use. On the other hand, ACT III is very fast at run-time, beating SCALP by 20 percent or so, depending on the particular mix of operations.

One big difference is that ACT III compiles subroutine jumps in machine language for floating point, rather than interpretive code. Typically these subroutine jumps are six instructions long. Consider two examples:

Example 1:

ACT III

```
a'+'b';'c''
b   a
h   addtemp
b   b
r   add
u
h   c
```

6

SCALP

```
c':='a'+'b';'
bf   a
af   b
hf   c
```

3

Example 2:

ACT III

```
a'+'b'+'c'+'d;'e''
b   a
h   addtemp
b   b
r   add
u
h   t1
b   t1
h   add temp
b   c
r   add
u
h   t2
b   t2
h   addtemp
b   d
r   add
u
h   e
```

18

SCALP

```
e':='a'+'b'+'c'+'d';'
bf   a
af   b
af   c
af   d
hf   e
```

5

These examples show how (1) ACT III uses up memory more quickly than does SCALP, and (2) in so doing achieves faster run times by jumping to special subroutines only when needed. The integer, subscript, and looping arithmetic in ACT III is all done in machine language.

On the compile side, SCALP wins out by a factor of almost 2 to 1. The double stack compiling scheme is extremely fast, but the chief reason is that SCALP uses a symbol table based on threaded lists. Briefly, each incoming symbol is placed in one of the 64 equivalence classes, and then the search is made within the class. Most of the time the search has to go only 2 or 3 elements down the sublist. Practically all the SCALP symbols are found immediately at the first level of the sublist. Another feature is that the symbol table reset procedure is part of the system in SCALP, but must be separately loaded in ACT III (tape T*).

In the area of debugging, SCALP both wins and loses. SCALP has very simple and effective trace, execute, and patch options that are unmatched in other systems. On the debit side, an error at compile time means recutting the source tape and starting over. In ACT III one can easily switch to typewriter and insert the correct statement by hand, then adjust the source tape in the photoreader and continue the compilation.

In the area of subroutines, SCALP easily and automatically loads only those subroutines needed, but give no flexibility for including new subroutines. ACT III requires that all subroutines that are a part of a given system be loaded, whether they are needed or not. However, ACT III does permit the user to prepare with SPAR his own special system including only those subroutines desired, though this process is very complicated.

## 7.6 Subroutines and the Library Tape

The library tape contains ten subroutines

integer △ integer
ln
real △ real (or integer △ real)
exp
real △ integer
sin-cos
arctan
sqrt
entier
random.

All subroutines contain a multiple of six instructions. A code word at the start of the subroutine gives the number of instructions and the address in the interpreter through which the subroutine is reached. The loader checks a ten bit code word. If the subroutine is needed, it is loaded into the first available space in memory. If it is not needed, it is bypassed by a series of six i0000 orders, repeated until the next subroutine is reached.

The real△real uses both ln and exp, and all three are loaded if needed. Or exp or ln may be loaded separately if they alone are needed. Sin and cos use the same subroutine, with a flag word on entry telling the difference. Sqrt operates by following a good starting approximation with two iterations in fixed point. ramdom uses an untested junk calculation that seems to give numbers having a resemblance to pseudo random numbers in the range $0 \leq x < 1$. Subroutines ln, exp, sin-cos, and arctan use Hastings approximations.

## 7.7 Some Remarks on Efficiency

The running time for equivalent programs can differ widely, and in ways that depend on the particular computer or system. Two features of SCALP, permitting mixed expressions and full subscript freedom, should be considered if running time is crucial. A third feature common to practically all compilers should also be considered.

1. The SCALP compiler inserts float operations as needed to permit mixed expressions, Thus, if x is real, 'x':=''1''+'x';' is compiled into: bi integer constant 1

float
af x
hf x

A more efficient construction is to make constants real if they are used in real expressions. Thus, 'x':=''1.''+'x';' would be compiled as:

bf real constant 1.0
af x
hf x

(Incidentally, ''1.'' is a legal representation of 1.0 in SCALP.) It is possible to have a compiler perform such floating of constants at compile time, and this should be recommended for the future.

2. The long time needed to compute subscripts dictates that the number of subscripted variables encountered at run time should be minimized. Thus, instead of

```
k':=''0'';'c'['i',,'j']':=''0.0'';'
for'k':='k'+''1''while'k'lte'n'do'
c'['i',,'j']':='c'['i',,'j']'+'a'['i',,'k']'*'b'['k',,'j']';'
```

as the familiar inner loop of a matrix multiply, one might use

```
k':=''0'';'sum':=''0.0'';'
for'k':='k'+''1''while'k'lte'n'do'
sum':='sum'+'a'['i',,'k']'*'b['k',,'j']';'
c'['i',,'j']':='sum';'
```

3. As is true in most present day compilers including SCALP, the compiler does not evaluate constant subexpressions at compile time. Therefore, if you do all your constant calculations in a preamble to your program, the running time will be lowered. For instance, instead of using

''2''*''3.141'592''

in the body of your program, replace this by

'twopi'

which is evaluated by: twopi':='2''*''3.141'592'';'

at the start of the program. Or you could calculate such a simple expression by hand. It should be recommended that future compilers perform the constant arithmmetic at compile time is such obvious cases.