A cross compiler for pocket calculators

Deane B. Blazie* and L. S. Levy†

A cross compiler producing programs for the HP-65 from source programs in BASIC is described. Optimisation techniques are discussed in detail, and typical programs presented showing the resultant code. Typical compilation costs are cited.

(Received November 1975)

In November of 1972, the first scientific pocket calculator was introduced by Hewlett Packard. The model HP-35 has a full complement of scientific functions, a storage register and an operational stack, all in a tiny package which easily fits into a shirt pocket. The introduction of the HP-35 marked the beginning of a major thrust in calculator technology.

Since then, hundreds of pocket calculators have appeared on the market, some with marked improvements over the HP-35. The technology advanced with improvements such as multiple storage registers, extended instructions and finally, programability. While all these factors expanded the market for pocket calculators, many purchasers of the programmable machines know little or nothing about programming, but are willing to learn enough about the discipline to program their calculators to solve problems.

To program a typical pocket calculator, one must press the appropriate keystrokes necessary to solve the problem, while the calculator is in the program mode. These keystrokes are equivalent to those needed to solve the problem by hand the first time. The keystrokes are machine instructions and as such are primitive. Programming these keystrokes can be quite difficult to one not well versed in the art of programming. It has been demonstrated, Dahl, Dijkstra and Hoare (1972) and is a general fact that unless programming is done in a systematic manner it can be difficult and frustrating. Carberry, Kahlil, Leathrum and Levy (1976) devote a chapter of their recent book *General Computer Science* to programming pocket calculators. In it they describe a systematic approach to programming and give programming examples.

The authors have taken this systematic approach a step further by allowing programs to be written in a high level language, BASIC, and then translated it into the calculator language by a computer program, a compiler. Actually, the computer program which accomplishes this translation is run on a medium or large scale computer and is therefore called a cross compiler.

The cross compiler thus reduces the problem of programming a pocket calculator to programming in a high level language. This reduces the programming effort considerably and because of the optimisation techniques used in the compiler, the calculator code generated is as efficient as that normally produced by hand.

The object code produced is executable on the model HP-65 calculator. The HP-65 was chosen because to date it is probably the most advanced programmable pocket calculator on the market. Furthermore, the Hewlett Packard programmable pocket calculators are the only ones offering conditional branching which is necessary in solving many problems.

1. Description of the HP-65

The HP-65 is a pocket size (1″ × 3″ × 6″) programmable computer. It can be used in the manual mode as a sophisticated scientific calculator or in the automatic mode where it can execute a stored sequence of keystrokes. The difference between an automatically run program and a sequence of keyboard activations is that the automatic program runs at high speed, and eliminates potential errors. Once a program is entered into the HP-65 program memory it can be loaded onto a $\frac{1}{2}$″ × 3″ magnetic card for storage and subsequently reloaded into the HP-65.

The HP-65 has a 100 step program memory. Most keyboard instructions occupy one memory location, but a few require two. Variable storage consists of an operational stack of four registers called the X, Y, Z and T registers and nine general purpose storage registers, $R1$ through $R9$. All arithmetic operations are performed in the stack registers. Unary operations ($1/X$, $\sqrt{X}$, etc.) are performed on the X register (the displayed register). Binary operations ($+$, $-$, X, Y/X, etc.) are performed using the X and Y register contents in the order $\mathbf{Y}\,\theta\,\mathbf{X}$ where θ is the binary operation. The stack is automatically pushed as a value is put into the X register and automatically popped when binary operations are performed. The stack can also be manipulated manually by any of several stack instructions. The entire stack may be cleared by performing a stack clear operation. The X-Y interchange operation effectively swaps the X and Y registers. The ENTER key performs a stack push operation, while two roll keys permit the stack to be rolled up or down to relocate its elements. The stack is explained in more detail in the discussion on optimisation.

The control structures of the HP-65 calculator include sequential control, conditional and unconditional branching and nonrecursive subroutine calls. Program steps may be labelled 0, 1, . . ., 9 or A, B, C, D, E. A, B, C, D, E are also subroutine names. A call to the subroutine labelled A causes control to pass to the instruction labelled A with sequential control from A until a RTN instruction is encountered. Control then returns to the instruction immediately following the call to A.

Any label can be used in a conditional or unconditional branch instruction. GTO 3 causes a transfer of control to the instruction labelled 3. Conditional branches can occur on the predicates $X \neq Y$, $\mathbf{X} \leq \mathbf{Y}$, $\mathbf{X} = \mathbf{Y}$, or $\mathbf{X} > \mathbf{Y}$, where X and Y are the stack registers mentioned earlier. When the predicate of a conditional instruction is true, the next two program steps are executed, otherwise they are skipped and program execution resumes with the third step after the predicate.

2. General description of the compiler

The compiler was designed with efficiency, ease of use, expandability, adaptability and transportability as primary goals. With only 100 program steps available, the compiled code

*US Army Human Engineering Laboratory, Aberdeen Proving Ground, Maryland, 21005, USA.
†Department of Statistics and Computer Science, University of Delaware, Newark, Delaware, 19711, USA.

must use this memory efficiently. Since programmable calculators are being bought by people who are not necessarily programmers, the compiler should be easy to use. Since few compilers survive without changes to improve their performance, steps were taken in the design to facilitate expanding the compiler. As new and different calculators are introduced, the compiler should be easily adaptable to produce their new machine code. Transportability, as used in this paper, refers to the ease with which the compiler can be run at computing centres using different machines.

The authors feel that most of these goals were best met with the structure shown in **Fig. 1.** This structure separates the lexical analyser, statement recogniser and semantic routines. Once a statement type is recognised, a semantic routine is called to compile codes for this statement. Each statement type has a unique syntax directed compiling routine which produces the output code for that statement type.

This structure facilitates the expandability and adaptability goals. To expand the source language only requires expanding the recogniser and adding an appropriate semantic routine. To adapt the compiler to a different calculator requires changes to the semantic routines, but the lexical analyser and recogniser need not be changed.

BASIC was chosen as the source language because BASIC is easy to learn and easy to use. Furthermore, BASIC is taught in most colleges and universities in introductory courses on computers. BASIC does, however, lack some needed capabilities that the calculator can perform. These shortcomings were remedied by adding some new constructs to the language. They are:

'DISPLAY'	Allows an expression to be computed and the calculator stopped while the evaluated expression is in the calculator display.
'DEGREES'	The HP-65 has a full complement of trigonometric functions requiring angular input. This input can be expressed in degrees, grads, or radians. The 'DEGREES' statement sets the mode to degrees.
'RADIANS'	Similar to degrees, this statement sets the angular mode to radians.
'GRADS'	Sets the angular mode to grads.
'INITIALISE'	Sets all variables in the program to zero.

Transportability could best be attained by writing the compiler in a language which was somewhat standardised and widely available. Whilst ANSI standard FORTRAN would have been nearly ideal in this respect, it was felt that the lack of string capabilities in FORTRAN would have severely compromised the compiler's operation in other respects. Therefore, PL/I was chosen as the language in which to implement the compiler. PL/I is widely available and somewhat standardised. The authors were careful not to use PL/I constructs which were known to be peculiar to a specific machine.

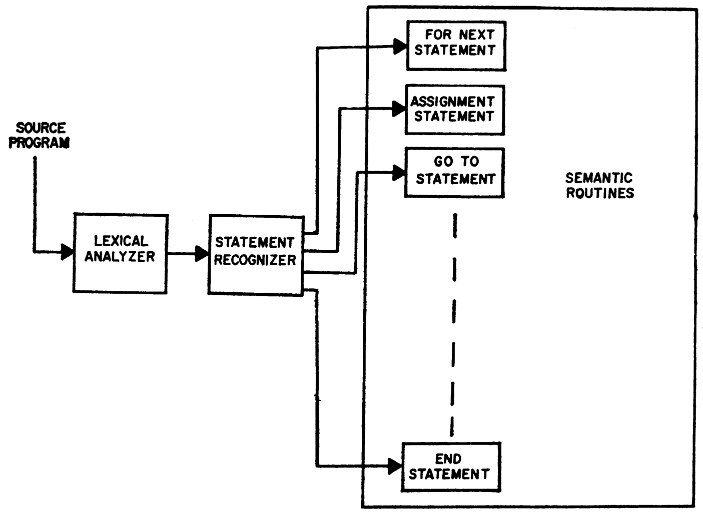


Fig. 1 Structure of the HP-65 cross compiler

The unique feature of this compiler is its means of achieving efficiency. The compiler structure, lexical analyser, and recogniser are covered in detail in the referenced literature, and will not be discussed here. The optimisation techniques, which account for the compiler's efficiency, are, however, of interest and will be discussed in detail in the remainder of this paper.

3. Optimisation

In writing the cross compiler for the HP-65, three primary limitations of the machine were considered.

1. Limited data storage (nine registers).
2. Limited program memory (100 program steps).
3. Limited stack size (four registers).

With only nine data storage registers available, a programmer must be careful not to use the registers wastefully, and to reuse a register whenever possible. To save program steps, a programmer should check to see if a value needed to evaluate an expression is already in the stack where it may be advantageous to use it. If a value in a register is just updated (that is, a recall, an operation, and a store), the register arithmetic feature of the HP-65 can be used.

These are optimisation techniques considered when one is programming by hand, and they should also be considered by the compiler if efficient code is to be produced. The compiler must also assure it does not over extend the machine by, for example, assigning more than nine registers or by assigning too many labels. An even more subtle precaution arises from the stack limit of four. The HP-65 evaluates expressions in reverse Polish form and automatically pushes and pops the stack as required. However, an arbitrary expression may not fit into the limited stack of four. If this happens, a programmer will usually break up the expression into intermediate results, and use a register to store these results while the rest of the expression is evaluated. Finally, the intermediate results are combined to obtain the evaluated expression. Alternatively, Carberry, Kahlil, Leathrum and Levy (1976) show how an expression can be manipulated in such a way that it makes more efficient use of the four register stack without using intermediate storage. This technique is used in the cross compiler in a limited, but advantageous, way to increase code efficiency.

4. Register and storage optimisation

The cross compiler makes efficient use of registers and stack space by modelling the nine general purpose registers and the four stack registers, keeping track of what is in each. For example, as variables are encountered in the source program, a register is assigned to each variable. Consider the statement in the BASIC language:

```
20 A = 3·5 * SIN (37·5) .
```

It computes a value for *A* by executing the HP-65 code:

```
3·5
ENTER
37·5
SIN
*
```

The result will now be in the *X* register. Then the compiler will search for an empty register and execute a STO *Ra*, where *Ra* is the empty register now assigned to the variable *A*. Now if the statement:

```
30 B = A * 2·715
```

is executed, the compiler will recognise that the value for *A* is in the X register; so it will not recall it, but will output the following code:

```
2·715
*
STO Rb
```

where Rb is the register assigned to B.

This is accomplished by maintaining the arrays:

```
REGISTER (9)
STACK (4)
REGBACKUP (9)
STKBACKUP (4)
STKFREZ (4)
```

REGISTER (I) is equal to zero if register I is not in use, or equal to the token associated with the variable using register I.

STACK (I) is similar, but represents the X, Y, Z and T registers for $I = 1, 2, 3$ and 4, respectively.

REGBACKUP (I) is a flag which is set when the variable using register I is also in the stack. That is, it is backed up in the stack. If all nine registers are in use and one more is needed, it may be possible to use a register, the contents of which are backed up in the stack.

STKBACKUP is similar; it is needed when a variable must be pushed through the bottom of the stack and then lost. However, if it is available in a register, there is no need to save it, and no sacrifice in losing it.

STKFREZ (I) is an array used when an expression is evaluated. A given expression is converted to reverse Polish form and evaluated on the stack by pushing variables onto the stack and operating on them. When a variable is needed, it is looked for in the X and Y stack registers first and accessed there, rather than calling it up from a storage register. However, as the variables are pushed onto the stack, they must not be disturbed until the expression has been evaluated; disturbing them would change their order, and the expression would be evaluated incorrectly.

As an example of the use of STKFREZ, consider the two-line program:

```
10 A = 2
20 B = A * 5
```

The compiled code will look like this:

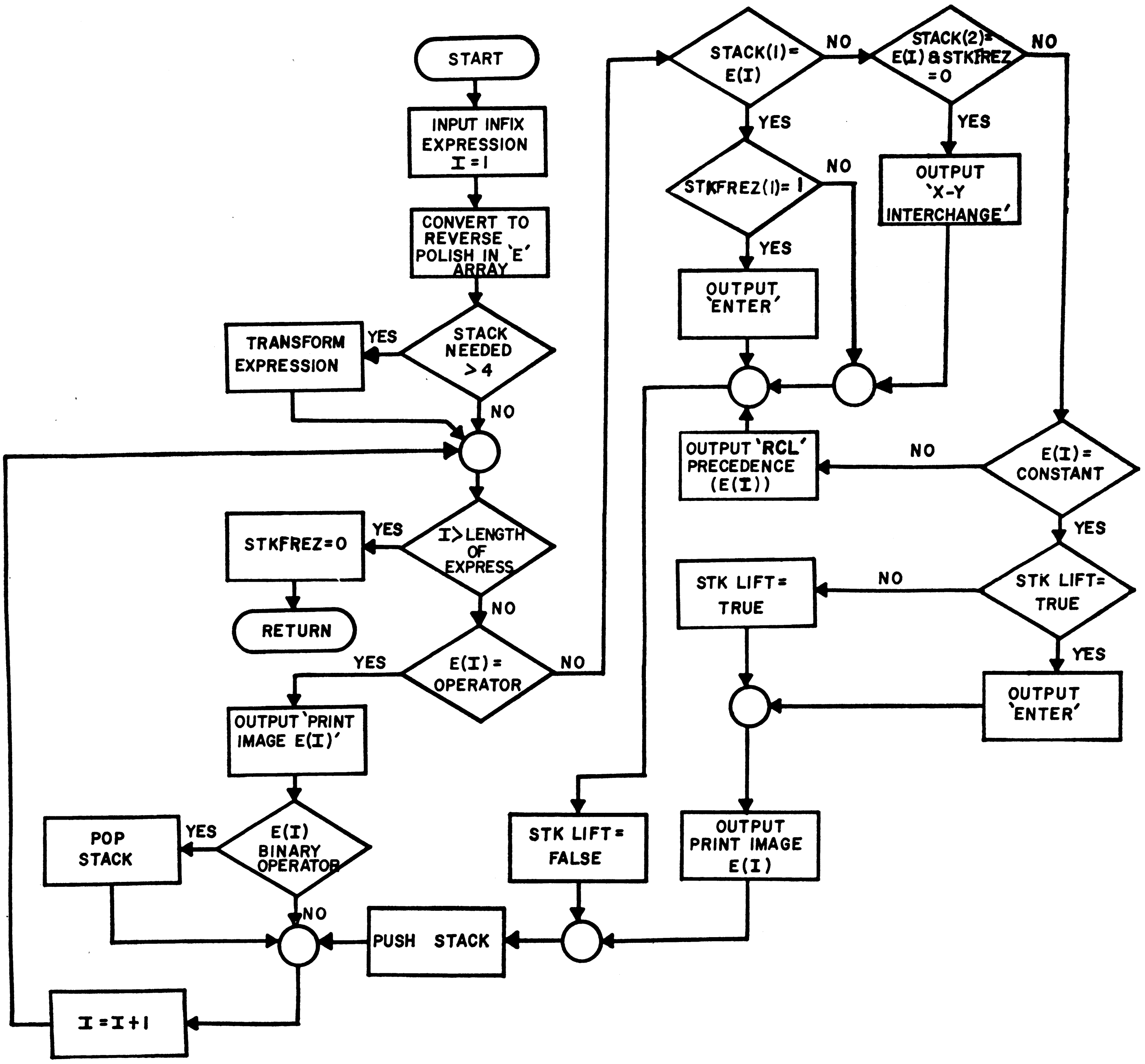


Fig. 2 Algorithm to evaluate expressions.

```
2       Puts 2 into the X register
STO 1   Stores X register into register 1
5       Puts 5 into the X register and moves 2 up into the
        Y register
*       Multiplies A by 5
STO 2   Stores B in register 2.
```

In the third step, the compiler examines the array STACK and finds A already in the X register, so it goes on to enter 5 and multiply.

Now consider a modification to this program:

```
10 A = 2
20 B = A * A
```

The compiled code for this program would be;

```
2
STO 1
ENTER
*
STO 2
```

Now at the third step, A will be in the X register, but the next operand is also A and—though it is already in the X register—it is needed there and must be pushed into the Y register by the next operand. That is, the first A is frozen; must not be removed. The array STKFREZ is used for this purpose. When an expression is evaluated, each time an operand is pushed onto the stack, STKFREZ (1) is set to indicate that that operand is frozen. Thus in step 3 of the target code above, the compiler knows that A is in the X register; but it also knows that STKFREZ (1) is set, and consequently, that it must perform an ENTER operation to enter A in both the X and Y registers.

The algorithm used to optimise the expression evaluation, as shown in the example, is flowcharted in **Fig. 2.** The following paragraphs will explain how it operates. Initially, all variables are set to zero. The algorithm then starts with step 1.

1. Transform the expression in infix form to an equivalent expression in reverse Polish form (postfix), and place the tokens in a linear array $E(I)$, where the evaluation will begin with the smallest I first. If the expression requires more than four stack levels, transform it (see *Stack level optimisation*) before proceeding with Step 2. $I = 1$.
2. If $E(I)$ is an operator, go to step 7; otherwise continue with step 3.
3. If STACK (1) $\neq E(I)$, then go to step 4; otherwise, if STKFREZ (1) = 0, go to step 8, or else output 'ENTER' and then go to step 8.
4. If STACK (2) $\neq E(I)$ or STKFREZ (2) $\neq 0$, go to step 5; or else output 'X-Y Interchange' and go to step 8.
5. If $E(I)$ is of type constant, then go to step 6; or else the value must be recalled from a register. Output the code 'RCL' and the value of the variable PRECEDENCE ($E(I)$). PRECEDENCE (token) contains the register number where the value of the token is stored.
6. If STKLIFT = .TRUE. then output 'ENTER' and output the print image of the constant. Set STKLIFT = .TRUE. and go to step 9.
7. Output the operator corresponding to the token $E(I)$. If $E(I)$ is a binary operator, then perform a stack POP operation. In either case set $I = I + 1$ and, if I is greater than the length of the expression, go to step 10; otherwise go to step 2.
8. STKLIFT = .FALSE. (stack does not need lifting).
9. Perform a stack PUSH operation, set $I = I + 1$, and go to step 2.
10. STKFREZ = 0 (unfreeze the entire stack). The evaluated expression is in the X register.

A stack PUSH is defined by:

⟨var⟩ (4) = ⟨var⟩ (3)
⟨var⟩ (3) = ⟨var⟩ (2)
⟨var⟩ (2) = ⟨var⟩ (1)

where ⟨var⟩ ::= STACK, STKBACKUP, STKFREZ and

STACK (1) = $E(I)$
STKFREZ = 1

Similarly, a stack POP is defined as:

⟨var⟩ (2) = ⟨var⟩ (3)
⟨var⟩ (3) = ⟨var⟩ (4)
STACK (1) = −STACK (1) (to denote an intermediate result)
STKBACKUP (1) = −STKBACKUP (1)
STKFREZ (1) = 1

where ⟨var⟩ is defined as above.

With the expression evaluated, the X register contains the numerical result. If the compiler is processing a DISPLAY statement, or part of a FOR statement, the result need not be stored; however, if it is compiling an assignment statement, the result must be stored in a register.

The algorithm which performs the register assignment is flowcharted in **Fig. 3.** The following three steps explain its operation.

1. If PRECEDENCE (⟨var⟩) = 0, then the variable has been previously assigned; go to step 2. Otherwise CALL FETCH REGISTER, a procedure which finds an available register and assigns it to the variable.
2. Output 'STO' and 'PRECEDENCE (⟨var⟩)' to store the value of the variable in the assigned register.
3. Perform the following operations:
   REGISTER(PRECEDENCE(⟨var⟩)) = var
   REGBACKUP(PRECEDENCE(⟨var⟩)) = 0 (the register contents are also available in the stack).
   STACK (1) = var
   STKBACKUP (1) = 0 (the X register contents are also saved in a register).

The fetch register algorithm finds an empty register if there is one, or 'steals' one if all registers are in use. The procedure, shown in **Fig. 4,** works in the following way:

1. Look at all nine elements of REGISTER(I). If one is zero,

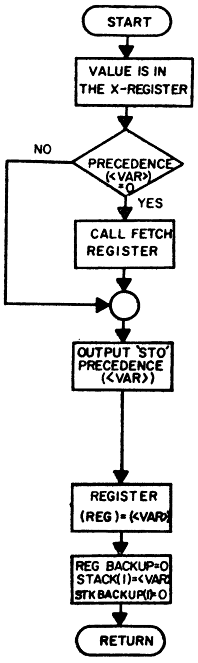


Fig. 3 Algorithm to save a variable in an assignment statement

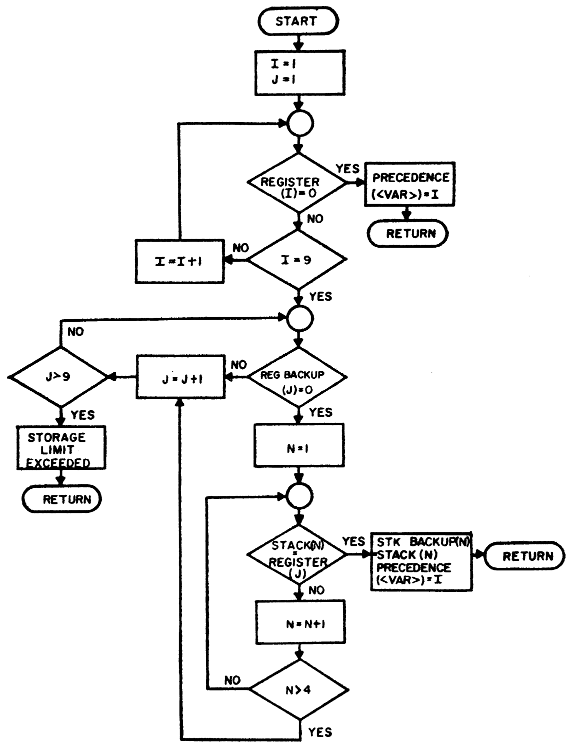


Fig. 4 Algorithm to get an available register.

then return this register number. Otherwise perform step 2.

2. For $I = 1$ to 9 find a REGBACKUP(I) = 0 such that, for some N, STACK(N) = REGISTER(I). This I is a register whose contents are in the stack element N. Set STKBACKUP(N) = STACK(N) to indicate that this stack element is no longer backed up in a register. Return I as the register available. If step 2 fails, the program has exceeded the storage capability of the calculator; and output an error message.

5. Stack level optimisation

As noted earlier in this paper, the HP-65 has a four register stack for evaluating arithmetic expressions in reverse Polish form. The expression

$$3 * 4 \text{ EXP } (5 + 6)$$

would be evaluated by the following HP-65 keystrokes.

3	Put 3 in the X register
ENTER	Push stack operation (see above)
4	Put 4 in the X register
ENTER	Push stack
5	Put 5 on the stack
ENTER	Push stack
6	Put 6 on the stack
+	Add X to Y (register contents) and pop stack
EXP	Raise X to Y power
*	Multiply X by Y

At the point just after 6 is entered from the keyboard, the stack contains:

T	3
Z	4
Y	5
X	6

Executing a '+' operation leaves the stack as:

T	3
Z	3
Y	4
X	11

The EXP operation produces:

T	3
Z	3
Y	3
X	4 EXP (11)

And the last operation '*' leaves:

T	3
Z	3
Y	3
X	3 * 4 EXP (11)

Now consider the expression:

$$3 * 4 \text{ EXP } (5 + 6 * 7) \ .$$

And in reverse Polish:

$$3\,4\,5\,6\,7 * + \text{ EXP } *$$

One may write the HP-65 code as:

```
3
ENTER
4
ENTER
5
ENTER
6
ENTER
7
*
+
EXP
*
```

Note the problem encountered when the fourth 'ENTER' command is performed. The stack was already full with four entries, and an ENTER would cause the top element, '3' in the T register, to be lost because the stack is only four deep.

One solution to this problem is fractionation: calculating part of the expression, storing the results, then calculating the rest of the expression, and combining the results. In the example, we could eliminate the necessity for intermediate storage by computing the equivalent expression

$$(4 \text{ EXP } (5 + 6 * 7)) * 3 \ .$$

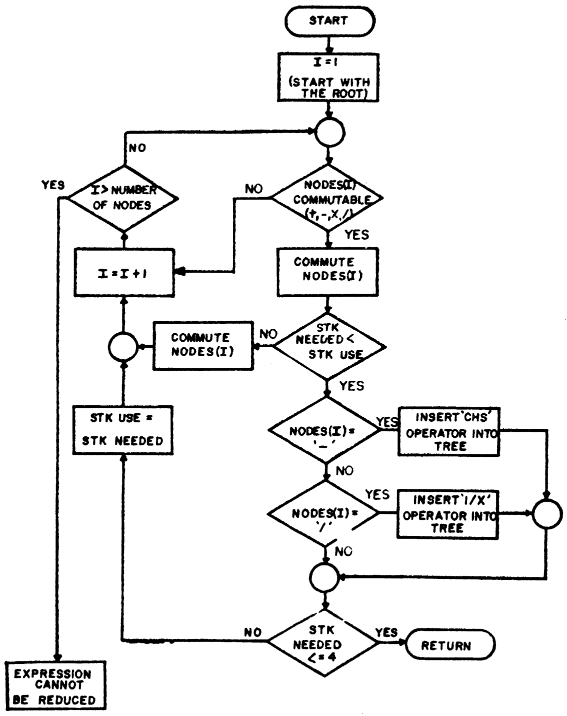
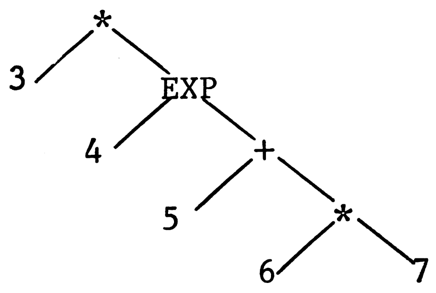


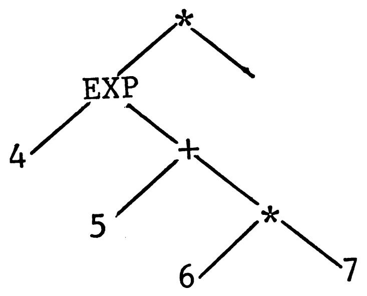
Fig. 5 Algorithm to reduce expression.

During the process of commuting the multiplication operation, we have reduced the required stack level to four.

This technique is employed in the cross compiler to reduce an expression so that a four level stack can store it. This procedure can best be explained by looking at the expression in tree form. The example just cited can be drawn as the following tree:



The cross compiler's algorithm for commute operations first commutes the root of the tree to get:



Evaluating this new tree requires a stack of level 4. Had the new tree not improved the stack efficiency, the old tree would have been reconstructed and the next commutable operation down the tree would have been commuted. The EXP node could not be commuted because exponentiation is not commutable, so the compiler would move down the tree until a commutable node is found.

While division and subtraction are not commutable operations, the expression can be modified to allow them to be commuted by inserting unary operators of the proper type that are available on the HP-65. For example the expression:

$$A - B$$

can be written as:

$$-B + A$$

where the unary minus operator is CHS, or 'change sign', on the HP-65. Similarly the expression:

$$A/B$$

can be written as:

$$(1/B) * A$$

where $1/B$ can be performed by the unary operation '$1/X$' on the HP-65. Thus the binary operations of addition, subtraction, multiplication and division can be commuted in attempts to reduce expressions to a stack level of 4.

The algorithm just discussed is shown in **Fig. 5.** It involves the following six steps:

1. Convert the expression into tree form, with the array NODES containing pointers to the nodes of the tree. NODES(1) points to the root. STKUSE is the level of stack required.
2. $I = 1$
3. If NODES(I) is a commutable node ($+$, $-$, $*$, $/$), then commute it. Otherwise go to step 5.
4. Check stack needed. If it is less than STKUSE, go to step 6. Otherwise commute NODE(I) again to restore it to its previous form.
5. $I = I + 1$. If I is greater than the last node, the tree cannot be reduced, return and output an error message. Otherwise go to step 3.
6. If the commuted node was subtraction or division, insert unary operator 'CHS' or '$1/X$' into proper branch of the expression tree. Go to step 5.

Expressions can be written which require arbitrary levels of stack use, even after applying this technique. Nevertheless, many practical examples have been tested, and none were flagged by the compiler as irreducible.

5. Performance, goals and achievements

Repeating the goals outlined in the design philosophy, the compiler was designed to be:

1. Efficient
2. Easy to use
3. Expandable
4. Adaptable to other machines
5. Transportable

Performance in terms of these goals is of interest.

Efficiency:
Techniques to insure efficiency were a major topic in this paper. Tests with a variety of sample problems establish that the compiler is quite efficient. Problem 2 was taken directly from the Hewlett Packard HP-65 STANDARD PAC. Designated STD-02A in the STANDARD PAC, it calculates basic statistics of a set of data points. The program in the STANDARD PAC requires 81 program memory locations, while the equivalent code generated by the compiler requires only 79 locations. This difference, though slight, is in favour of the compiler. It is quite surprising because, obviously, a skilled programmer could generate code as efficiently as the compiler—or even more efficiently, since he could make numerous optimisation passes over the code. However, the compiler can produce consistently good code in much less time, and the option still exists for a programmer to optimise the compiled code further. The cost of running the compiler for the same problem on the Burroughs B6700 computer at the University of Delaware Computing Centre was 26 cents. While the cost of generating the code by hand is not known, the author feels that ten times as long is a very conservative estimate.

Ease of use:
No attempt was made to compare the ease with which programs can be generated using the compiler versus generating the code by hand, but few would argue that programming in BASIC is at all comparable to the difficulty of programming in machine language. Certainly the number of compilers in use today substantiates the fact that programming in a high level language is much easier than programming at the machine level.

Expandability:
Since the compiler was designed and implemented, only one additional source language statement was added. Early in the testing phase, it became apparent that the use of the register and the stack clear operations were not available to the user, and that users needed a construct to make them available. The INITIALISE statement, which does this, was added without difficulty; it merely required additions to the compiler, with no changes or deletions in other semantic routines. The addition of this new construct established the ease with which the compiler can be expanded.

Adaptability to other machines:
No attempt has been made to adapt the compiler to a machine other than the HP-65, but this adaptation would only require changing the semantic routines.

Transportability:
It was mentioned earlier that the compiler was implemented in a standard subset of PL/I so it may be easily moved to other machines. While the PL/I program has, to date, not been transported to any other machine, the design was transferred into an HP-9830 calculator in the language BASIC.

While this does not substantiate total portability to all machines, it does show that the design is transportable to another language and, hence, to other machines.

Thus far, the compiler design philosophy has been described, the block diagram of its structure given, and the optimisation techniques used on the output code explained in some detail. We now wish to give some examples of problems which have been solved using the compiler.

The appendix contains the results of the sample problems, in the order in which they are presented below.

Problem 1
Consider the problem of finding the *n*th Fibonacci number. The Fibonacci series is defined by the recurrence equations:

$$X_{n+1} = X_n + X_{n-1} \ .$$

The results in the Appendix give the source and compiled codes to solve the problem.

Problem 2
This problem is program number STD02A in the Hewlett Packard STANDARD PAC library which is given with every calculator: 'MEAN, STANDARD DEVIATION, STANDARD ERROR'. A BASIC program to solve the problem, and the solution, are given in the Appendix. The program uses the 'INITIALISE' statement to generate the first two machine instructions, 'CLR STK' and 'CLR REGS'. The example also points out the use of register arithmetic in the output code, as evidenced in instructions 11 through 13, the compiled codes for the statement $S = S + A$. As noted above, the code takes up two less memory locations than the manufacturer's solution.

Problem 3
This problem solves the quadratic equation:

$$aX^2 + bX + c = 0 \ .$$

The *A* button allows the user to input the constants *a*, *b* and *c*, then displays the discriminant *D*. If *D* is less than 0, the *C* button gives the complex roots, otherwise the *B* button gives the real roots.

Problem 4
As an example of solving a general recurrence equation (by iteration since the HP-65 does not allow recursive subroutine calls) the solution for:

$$X_{n+1} = P(n)X_n + Q(n)X_{n-1}$$
$$\text{where } X_0 \text{ and } X_1 \text{ are given} \ ,$$

is programmed for:

$$X0 = 0;\ X1 = 1;$$
$$P(n) = n + 1$$
$$Q(n) = 2n$$

P and *Q* are programmed in subroutines *A* and *B* and can be changed to any function the user wishes. The program with *P* and *Q* defined as above requires 59 program memory locations.

Appendix

Sample problem 1 Fibonacci number problem

```
 10 INPUT N
 20 X1 = 0
 30 X2 = 1
 40 N1 = 3
 50 X3 = X1 + X2
 60 IF N = N1 THEN 110
 70 X1 = X2
 80 X2 = X3
 90 N1 = N1 + 1
100 GOTO 50
110 DISP X3
120 END
```

Source problem

```
 1  R/S          INPUT N
 2  STO    1
 3  0
 4  STO    2
 5  1
 6  STO    3
 7  3
 8  STO    4
 9  LBL
10  2
11  RCL    2
12  RCL    3
13  +
14  STO    5
15  RCL    1
16  RCL    4
18  X = Y
19  GTO
20  1
21  RCL    3
22  STO    2
23  RCL    5
24  STO    3
25  1
26  STO
27  +
28    4
29  GTO
30  2
31  LBL
32  1
33  RCL    5
34  R/S
35  R/S
```

Object program

```
STORAGE MAP
VARIABLE  REGISTER
N            1
X1           2
X2           3
N1           4
X3           5
  35 PROGRAM STEPS USED
```

Sample problem 2 Mean, standard deviation, standard error

```
  1 INITIALISE
 10 SUBROUTINE A
 20 INPUT A
 30 S = S + A
 40 N = N + 1
 50 S2 = S2 + A * A
 60 GOTO 20
100 SUBROUTINE E
110 INPUT A
120 S = S − A
130 S2 = S2 − A * A
140 N = N − 1
150 GOTO 110
200 SUBROUTINE B
```

```
210 X = S/N
220 DISP X
300 SUBROUTINE C
310 S3 = SQRT((S2 − N * X * X)/(N − 1))
320 DISP S3
400 SUBROUTINE D
410 DISP S3/SQRT(N)
420 END
```

Source program

```
 2  CLR STK
 4  CLR REGS
 5  LBL
 6  A
 7  LBL
 8  1
 9  R/S                      INPUT A
10  STO          1
11  STO
12  +
13     2
14  1
15  STO
16  +
17    3
18  RCL          1
19  ENTER
20  *
21  STO
22  +
23      4
24  GTO
25  1
26  LBL
27  E
28  LBL
29  2
30  R/S                      INPUT A
31  STO          1
32  STO
33  *
34      2
35  RCL          1
36  ENTER
37  *
38  STO
39  *
40    4
41  1
42  STO
43  *
44      3
45  GTO
46  2
47  LBL
48  B
49  RCL          2
50  RCL          3
51  /
52  STO          5
53  R/S
54  LBL
55  C
56  RCL          4
57  RCL          3
58  RCL          5
59  *
60  RCL          5
61  *
62  *
63  RCL          3
64  1
65  *
66  /
68  SQRT
69  STO          6
70  R/S
71  LBL
72  D
73  RCL          6
74  RCL          3
76  SQRT
77  /
78  R/S
79  R/S
```

Object program

```
STORAGE MAP
VARIABLE        REGISTER
A                  1
S                  2
N                  3
S2                 4
X                  5
S3                 6
  79 PROGRAM STEPS USED
```

Sample problem 3 Quadratic equation solver

```
  5 SUBROUTINE A
 10 INPUT A
 20 INPUT B
 30 INPUT C
 50 D = (B * B − 4 * A * C)
 60 DISP D
 70 SUBROUTINE B
 80 DISP (−B + SQRT(D))/(2 * A)
 90 DISP (−B * SQRT(D))/(2 * A)
100 SUBROUTINE C
110 DISP −B/(2 * A)
140 END
```

Source problem

```
 1  LBL
 2  A
 3  R/S                INPUT A
 4  STO         1
 5  R/S                INPUT B
 6  STO         2
 7  R/S                INPUT C
 8  STO         3
 9  RCL         2
10  ENTER
11  *
12  4
13  RCL         1
14  *
15  RCL         3
16  *
17  −
18  STO         4
19  R/S
20  LBL
21  B
22  RCL         2
24  CHS
25  RCL         4
27  SQRT
28  +
29  2
30  RCL         1
31  *
32  /
33  R/S
34  RCL         2
36  CHS
37  RCL         4
39  SQRT
40  −
41  2
42  RCL         1
43  *
44  /
```

```
   45  R/S
   46  LBL
   47  C
   48  RCL
   50  CHS
   51  2
   52  RCL          1
   53  *
   54  /
   55  R/S
   56  RCL          4
   58  CHS
   60  SQRT
   61  2
   62  RCL          1
   63  *
   64  /
   65  R/S
   66  R/S
```

Object program

```
STORAGE MAP
VARIABLE        REGISTER
A                   1
B                   2
C                   3
D                   4
  66 PROGRAM STEPS USED
```

Sample problem 4   General recurrence equation

```
 10 INITIALISE
 15 INPUT N1
 20 X1 = 1
 30 N = 2
 40 GOSUB A
 50 GOSUB B
 60 X2 = P1 * X1 + P0 * X0
 60 X2 = P1 * X1 + P0 * X0
 70 IF N = N1 THEN 120
 80 X0 = X1
 90 X1 = X2
100 N = N + 1
120 DISP X2
110 GOTO 40
130 SUBROUTINE A
140 P1 = N * N + 1
150 RETURN
160 SUBROUTINE B
170 P0 = N * 2
180 RETURN
200 END
```

Source problem

```
 2  CLR STK
 4  CLR REGS
 5  R/S                  INPUT N1
 6  STO         1
 7  1
 8  STO         2
 9  2
10  STO         3
11  LBL
12  2
13  A
14  B
15  RCL         4
16  RCL         2
17  *
18  RCL         4
19  RCL         4
20  *
21  +
22  STO         4
23  X − Y
24  RCL         1
26  X = Y
27  GTO
28  1
29  RCL         2
30  STO         4
31  RCL         4
32  STO         2
33  1
34  STO
35  +
36    3
37  LBL
38  1
39  RCL         4
40  R/S
41  GTO
42  2
43  LBL
44  A
45  RCL         3
46  ENTER
47  *
48  1
49  +
50  STO         4
51  RTN
52  LBL
53  B
54  RCL         3
55  2
56  *
57  STO         4
58  RTN
59  R/S
```

Object program

```
STORAGE MAP
VARIABLE        REGISTER
N1                  1
X1                  2
N                   3
X2                  4
P1                  4
P0                  4
X0                  4
  59 PROGRAM STEPS USED
```

References

Aho, A. V., and Ullman, J. D. (1972). *The Theory of Parsing, Translating, and Compiling*, Vols. 1 and 2, Prentice-Hall.

Blazie, D. B. (1976). A Compiler for Pocket Calculators, Masters Thesis, University of Delaware, Department of Statistics and Computer Science.

Carberry, S., Kahlil, H., Leathrum, J. F., and Levy, L. S. (1976). *General Computer Science*, Merril & Co.

Cocke, J., and Schwartz, J. T. (1970). *Programming Languages and Their Compilers*, New York University.

Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R. (1972). *Structured Programming*, London, New York, Academic Press.

Elson, M. (1973). *Concepts of Programming Languages*, Science Research Associates, Inc.

Floyd, R. W. (1963). Syntactic Analysis and Operator Precedence, *JACM*, Vol. 10, pp. 316-333.

Gries, D. (1971). *Compiler Construction for Digital Computers*, John Wiley & Sons.

Hopgood, F. R. A. (1900). *Compiling Techniques*, London: MacDonald; New York: American Elsevier.

Lee, J. A. N. (1967). *The Anatomy of a Compiler*, Reinhold Publishing Co.