16. AN EXAMPLE OF RECORD PROCESSING—DIFFERENTIATING AN EXPRESSION

We now present, as an example of record processing, a program for the formal differentiation of expressions. We consider expressions such as $x^2 + 3x + 5$ or $(2x^3 + 1) \times (3x + 5)$, using one variable, x, and the operations of addition, multiplication, and exponentiation to a positive integer power. From this it should be easy to see how we could deal with expressions involving several variables and more operations. We do not discuss this extension but we set up the program in a general form which permits it. Formal differentiation takes an expression e and differentiates it to produce another expression $\frac{de}{dx}$, using the rules explained in elementary books on calculus. We recall that if e_1 and e_2 are expressions in x and n is a constant:

$$\frac{d}{dx}(e_1 + e_2) = \frac{d}{dx}(e_1) + \frac{d}{dx}(e_2) \tag{1}$$

$$\frac{d}{dx}(e_1 \times e_2) = e_2 \frac{d}{dx}(e_1) + e_1 \frac{d}{dx}(e_2) \tag{2}$$

$$\frac{d}{dx}(x^n) = n\ x^{n-1} \tag{3}$$

$$\frac{d}{dx}(n) = 0 \tag{4}$$

$$\frac{d}{dx}(x) = 1 \tag{5}$$

For example

$$\frac{d}{dx}((2x^3 + 1) \times (3x^2 + 5))$$

$$= (3x^2 + 5) \times \frac{d}{dx}(2x^3 + 1) + (2x^3 + 1) \times \frac{d}{dx}(3x^2 + 5)$$

$$= (3x^2 + 5) \times 6x^2 + (2x^3 + 1) \times 6x$$

Here is a suitable POP-2 program (it is followed by explanatory notes).

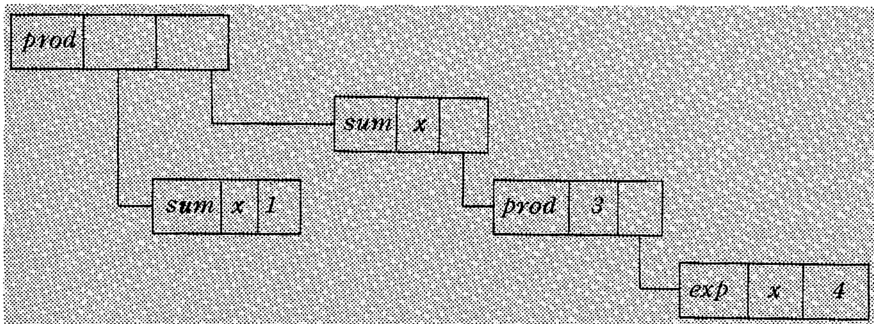
```
vars sum1 sum2 destsum operation 4 ++;
recordfns ("sum", [0 0]) -> sum1 -> sum2 -> destsum -> nonop ++;
vars prod1 prod2 destprod operation 3 **;
recordfns ("prod", [0 0]) -> prod1 -> prod2 -> destprod -> nonop **;
vars exp1 exp2 destexp operation 2 ↑↑;
recordfns ("exp", [0 0]) -> exp1 -> exp2 -> destexp -> nonop ↑↑;

function epr e; comment prints an expression;
    if e.isnumber or e.isword then pr(e)
        elseif e.dataword = "sum" then pr("("); epr(sum1(e)); pr("++");
                                        epr(sum2(e)); pr(")")
        elseif e.dataword = "prod" then epr(prod1(e)); pr("**");
                                        epr(prod2(e))
        elseif e.dataword = "exp" then epr(exp1(e)); pr("↑↑"); epr(exp2(e))
    close
end;
```

```
vars differror;
function diff e;
    if e.isnumber then 0
        elseif e.isword then if e = "x" then 1 else differror(e) close
        elseif e.dataword = "sum" then diff(sum1(e)) ++ diff(sum2(e))
        elseif e.dataword = "prod" then prod2(e) ** diff(prod1(e)) ++
                                        prod1(e) ** diff(prod2(e))
        elseif e.dataword = "exp" then exp2(e) ** exp1(e) ↑↑ (exp2(e)-1)
        else differror(e)
    close
end;
function differror(e); nl(1); pr([diff error]); epr(e) end;
```

We start by introducing three kinds of records, *sums*, *prods* and *exps*, each with two components, to represent the three ways of building symbolic expressions: by addition, multiplication, and exponentiation. For each we obtain two select/update function doublets, a destructor and a constructor. We assign the constructor function not to an ordinary variable but to one with an operation identifier with appropriate precedence. This enables us to construct symbolic expressions very easily. For example,

```
vars e;
("x" ++ 1) ** ("x" ++ 3 ** "x" ↑↑ 4) -> e;
```

constructs, and assigns to *e*, an expression which may be pictured as



Thus, for example, *dataword*(*e*) is "*prod*" and *sum1*(*prod1*(*e*)) is "*x*"
The function *epr* prints an expression

```
epr(e);
(x ++ 1) ** (x ++ 3 ** x ↑↑ 4)
```

The function *diff* differentiates an expression, first testing what kind of expression it is, and then, if it is complex, combining the components in the appropriate way, differentiating them where necessary, using a recursive application of *diff*.

```
eprint(diff(e));
(1 ++ 0) ** (1 ++ (3 ** 4 ** x↑↑3 ++ 0 ** x↑↑4))
```

This result is correct but cumbersome. It would be nice to have it simplified. Also it would be a good idea if expressions, like x↑↑x, which cannot be handled by the program, were rejected. We can accomplish both these aims by using more elaborate functions for ++, **, and ↑↑ than the simple record constructors. We could start all over again or simply carry on assigning new values to these variables. Let us do the latter.

```
vars conssum;nonop ++ -> conssum;
function makesum e1 e2;
    if e1 = 0 then e2
      elseif e2 = 0 then e1
      else conssum(e1, e2)
    close
end;
makesum -> nonop ++;
```

The previous value of ++, that is, the function to construct a *sum* record, has been saved in the variable *conssum*. The function *makesum* checks for the zero case and only calls *conssum* to construct a *sum* record if neither argument is zero. The function *makesum* is assigned as the new value of ++. To follow what is going on we must carefully distinguish between variables and the functions which are their values. Similarly

```
vars consprod consexp;nonop ** -> consprod;nonop ↑↑ -> consexp;
function makeprod e1 e2;
    if e1 = 0 or e2 = 0 then 0
      elseif e1 = 1 then e2
      elseif e2 = 1 then e1
      else consprod(e1, e2)
    close
end;
makeprod -> nonop **;
function makeexp e1 e2;
    if not(e1 = "x") or not(e2.isnumber) then differror(e1);differror(e2)
    elseif e2 = 0 then 1
    elseif e2 = 1 then e1
    else consexp(e1, e2)
    close
end;
makeexp -> nonop ↑↑;
```

More simplification could be done, for example, replacing $x+x$ by $2x$, but this is to some extent a matter of taste, and it is rather more difficult if we want to do things like replacing $x+3x^2+x$ by $2x+3x^2$.

It is worth remarking in closing that there are other ways of representing these expressions in POP-2, for example, by using arrays (see section 17) or lists, or by making a different use of records. They may be more advantageous in some ways, for example, brevity of program, speed of running, or economy of store space. For example, if we are restricting ourselves to polynomials in x we could represent $1+6x+5x^2$ by an array a with $a(1) = 1, a(2) = 6$ and $a(3) = 5$.

EXERCISES

1. Extend the differentiation program so that it deals with expressions in several variables and differentiates with respect to any one of them. Remember that differentiating v_1 by v_2 gives 0 unless v_1 and v_2 are the same variable.

2. Write a function *eval* such that *eval(e, n)* is the value of the expression *e* when "*x*" has the numerical value *n*. The expression *e* is to be restricted to the kind of expression accepted by the differentiation program. Use it and the differentiation program to write a function to differentiate a given expression *k* times, and tabulate the numerical value of the result from *a* to *b* in intervals of *delta*.

3. Let us use 'simple sentence' to mean any sequence of English words except 'not', 'and', 'or' or 'implies'. We define propositional expressions as follows:

(a) A simple sentence is a propositional expression.
(b) If p is a propositional expression so is *not*(p).
(c) If p_1 and p_2 are propositional expressions, so are p_1 *and* p_2, p_1 *or* p_2, and p_1 *implies* p_2.

Write a program which accepts a sequence of simple sentences and negations of simple sentences, and is then able to produce an answer *true*, *false*, or *unknown* when given any propositional expression. (Remember that p_1 *or* p_2 is true if one or both of p_1 *and* p_2 are true, and p_1 *implies* p_2 is true unless p_1 is true and p_2 is false.)

17. ARRAYS

An array is a table of items. It may be of one or more dimensions. Whereas each of the components of a record is accessed by name, the individual items of an array are indexed by number.

There is a standard function *newarray* which sets up an array with specified dimensions and initializes the items of the array. For example, the statement

newarray([*1 5 1 5*], **nonop** *) –> *a*

sets up a square array 5 by 5 and initializes each element $a(i, j)$, to the product of i and j. Thus a looks something like this:

		j				
		1	2	3	4	5
	1	1	2	3	4	5
	2	2	4	6	8	10
i	3	3	6	9	12	15
	4	4	8	12	16	20
	5	5	10	15	20	25

The first parameter of *newarray* must be a list of integers which alternately represent the lower and upper bounds of each dimension of the array. The second parameter must be a function which requires n arguments, where n is the number of dimensions of the array, and produces one result. The function is evaluated for every combination of subscripts and the result is inserted in the generated array.

The elements of the array can be accessed and updated as follows:

a(*3*, *4*) =>

prints the contents of row 3 column 4.

–*1* –> *a*(*1*, *1*);

replaces the contents of row 1 column 1 with –1. The array a is actually a doublet—a function with an updater part.

There are a number of important advantages of removing the distinction between arrays and functions. First, all the POP-2 facilities for handling functions can handle arrays equally well; secondly, it gives

the user a free choice of representation by rule or by table. Thus it is more economical to represent the matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

by a function:

```
function diag i j;
if i = j then 1 else 0 close
end
```

than by storage of the actual tables in the form of an array.

Consider the following function definition

```
function addm m1 m2 ni nj;
newarray ([% 1, ni, 1, nj %], lambda i j;
m1(i, j) + m2(i, j) end)
end
```

The function *addm* adds two *ni* by *nj* matrices *m*1 and *m*2 together by adding their corresponding elements. It will work equally well with two-parameter functions and with two-dimensional arrays. The function *addm* generates a new array, each element of which contains the sum of the corresponding elements of the original arrays or functions.

EXERCISES

1. Define a function to test for a winning position in noughts and crosses (tic-tac-toe). Represent the board by an array whose elements have as values the words *nought*, *cross*, or *blank*.

Define a further function to place a piece in a square to achieve, if possible, a winning position for that piece. If there is no winning position, the other player should be blocked from winning if possible.

2. Define a function to multiply two p by p matrices M and N using the definition $M \times N = L$ where $L_{ik} = \sum_j M_{ij} N_{jk}$.

3. Write a function to sort the elements of a one-dimensional array into ascending order. This can be done by going through the array looking at each pair of elements in turn and interchanging them if they are in the wrong order. The process is then repeated until a whole pass through the array produces no interchanges.

18. STRIPS

Although for many purposes POP-2 arrays prove adequate, there are situations where a more primitive system for storing information is needed. The use of triangular arrays or arrays consisting entirely of truth values are examples of situations where, although standard arrays are adequate, they are inefficient or wasteful. Such special arrays can be defined in terms of the more primitive data structures called *strips*.

A strip is a one-dimensional data-structure with a fixed number of components. As we will explain, the method of accessing the components of a strip is different from that for an array. All the components in a given strip must be the same size. For example, all the components of a given strip might be full items capable of storing any POP-2

value. By a component of size n we mean one of n bits, that is, in the range 0 to 2^{n-1}. Conventionally size 0 means a whole machine word, containing any integer, real, function, list, and so on. The *length* of a strip is the number of components in the strip. The standard POP-2 arrays described in the previous section can be defined in terms of strips of full items.

A *strip class* is a class of strips, each strip having the same component size but not necessarily the same number of components. There are two standard strip classes, known as *full strips* and *character strips*. Full strips have full items as components. The size of each component of a character strip is six. The components of a character strip can have any integer value in the range 0 to 63, and can thus be used to represent alphabetic or numeric characters in internal code.

There is a standard function *init* (initiate) such that *init*(n) is a new full strip with elements numbered 1 to n, all initially undefined, and a standard function *subscr* (subscript) such that *subscr*(i, s) is the ith element of the full strip s. For example

```
vars s; init(10) -> s;
13 -> subscr(3, s); subscr(3, s) =>
**13
```

Compare the second line with the equivalent statements for an array a

```
13 -> a(3); a(3) =>
```

The difference is that a is a function and s is a data structure. We could define the array a thus:

```
function a i; subscr(i, s) end;
lambda x i; x -> subscr(i, s) end -> updater(a);
```

In fact the standard function *newarray* produces arrays in this sort of way, but it can do it more economically by using the device of *partial application* described in the next section.

For character strips the standard initiating and subscripting functions are *initc* and *subscrc*.

The standard class of character strips, also called 'strings', is provided primarily to enable one to manipulate sequences of characters. We can read in such a sequence by enclosing the characters in string quotes, thus

```
"the cat sat on the mat.\\ -> s;
s =>
** "the cat sat on the mat.\\
```

The value of s is a character strip, and so if we write *subscr*(3, s) we obtain the integer between 0 and 63 which represents the character e. If *nb* is the integer representing the character b [see Appendix 1 of the Reference Manual (Part 3)], we could say

```
nb -> subscrc(5, s); s =>
** "the bat sat on the mat.\\
```

A new strip class can be defined using the standard function *stripfns*. The statement

```
stripfns("tstrip\\, 1) -> subscrt -> initt;
```

defines a new strip class with component size 1, that is, 1 bit, each component being 0 or 1. The word associated with the strip class is "*tstrip*\\. A function for constructing strips of this class is assigned

to the variable *initt* (short for 'initialize t-strip') and a function for accessing strips of the class is assigned to the variable *subscrt*. Thus a strip of truth values can be constructed by the statement

initt(30) —> x;

which constructs a strip x of 30 truth values. The 30 truth values are initially undefined. The *subscrt* function can be used to place truth values in the strip. For example, the statements

1 —> subscrt(4, x);
0 —> subscrt(2, x);

place the truth value 0 (false) in position 2 of x and the truth value 1 (true) in position 4 of x.

EXERCISES

1. Write a function to convert a list of characters to a string.

2. Write a function to concatenate two strings. Make it an operation called <—> of precedence 2. (The standard function *datalength* when applied to a strip gives the number of components in it.)

3. If s is a strip defined for $k = 1$ to 100, create a select/update doublet a to represent an array with elements $a(i,j)$, $i=1$ to 10, and $j=1$ to 10. If $a2(i,j) = 0$ for all $i<j$, create an array $a2$ using a strip of only 55 elements.

19. PARTIAL APPLICATION

When a function is executed, the values of the actual parameters are assigned to the formal parameters and the body of the function definition executed. *Partial application* enables one or more of the actual parameters to be assigned without executing the function. The result of partially applying a function to one or more parameters is always a function. This resulting function will be like the original function but will have fewer formal parameters. This is best shown by an example, as follows.

The function *tab* defined in an earlier section requires four parameters: a function and three integers defining the range. If a more specific version of *tab* which always tabulated up to 100 were required, we could produce this new function by partially applying *tab* to 100 and assigning the result to, say, *tab1*. Partial application is indicated by using *decorated parentheses* (% and %) in place of the plain parentheses. In addition, when the number of actual parameters is less than the number of formal parameters, it is always the rightmost of the formal parameters which are given values. Thus the statement

tab(% 100 %) —> tab1

makes *tab1* a function of three parameters. The three parameters of *tab1* correspond to the *first* three parameters of *tab*. *tab1* could then be called by executing, say

tab1(sqrt, 50, 10);

which would tabulate the values of $\sqrt{50}$, $\sqrt{60}$, and so on, up to $\sqrt{100}$.

Thus partial application enables us to start off by defining a very general function with many parameters and then specialize it to obtain one or more less general functions. Another example would be a

function *distance* defined as

```
function distance x y u v;
    sqrt((x—u) * (x—u) + (y—v) * (y—v))
end;
```

If we are particularly interested in distances from Edinburgh (coordinates —50, 350), London (—20, 10), and Birmingham (—70, 70), we define

```
vars disted distlo distbi;
distance(% —50, 350 %) —> disted; distance(% —20, 10 %) —> distlo;
distance(% —70, 70 %)  —> distbi;
```

We can use these functions thus:

```
disted(50, —50) =>
**316.0
```

Since arrays are functions, partial application can be used on them. If *a* is a two-dimensional array, *a* (% 3 %) is a one-dimensional array consisting of the third column of *a*. Thus

```
a(2, 3) =>
**7
a(% 3 %) —> b;
b(3) =>
**7
8 —> b(2);
```

We can now see how to create a one-dimensional array from a strip;

```
vars s a;
init(10) —> s; subscr(% s %) —> a;
33 —> a(3); a(3) =>
**33
```

How can we write a general function to produce a two-dimensional array from a strip, indexed by $i = 1$ to n, $j = 1$ to n?
Consider first

```
function array2 n => a; vars s;
    init(n*n) —> s;
    lambda i j; subscr(n*(i—1) + j, s) end —> a;
    lambda x i j; x —> subscr(n*(i—1) + j, s) end —> updater(a)
end;
```

if we test this by

```
vars a; array2(10) —> a;
99 —> a(3, 4); a(3, 4) =>
```

we get an error message saying that *n* is undefined, so is *s*. That is because we have created a doublet *a* for the array and this doublet is used *outside* the function *array2*, which has *n* and *s* as variables. As soon as *array2*(10) has been evaluated these variables are no longer in existence, that is, their values are not accessible any more. So when we say *99 —> a*(*3*, *4*), causing the lambda expression

```
lambda x i j; x —> subscr(n*(i—1) + j, s) end
```

to be entered with $x = 99$, $i = 3$, and $j = 4$, the attempts to evaluate *n* and *s* out of their proper context will cause an error.

How can we remedy this trouble? We would like to attach the values of *n* and *s* to the lambda expression so that wherever the lambda expression goes the values of *n* and *s* are sure to go too. We do this by making *n* and *s* into formal parameters and then partially applying

the lambda expression to the values we want them to have. What are these values? They are the values of *n* and *s* while *array2* is being executed. Thus we write

```
function array2 n => a; vars s;
      init(n * n) —> s;
      lambda i j n s; subscr(n* (i–1) + j, s) end (% n, s %)
          —> a;
      lambda x i j n s; x —> subscr(n *(i–1) + j, s) end (% n,s %)
          —> updater (a)
end;
```

When we test this it gives the correct answer.

```
vars a; array2(10) —> a;
99 —> a(3, 4); a(3, 4) =>
**99
```

To sum up, the trouble occurs if a function mentions some variables which are not formal parameters, output locals or locals (such variables are called non-local variables), and is then called when these variables are no longer in existence. It is remedied by making these variables formal parameters and partially applying to their values when the function is created, thus instead of

lambda *x y;* . . . . . . *a* . . . . *b* . . . . **end**
write
lambda *x y a b;* . . . . . . *a* . . . . *b* . . . . **end** (% *a*, *b* %)

Instead of
function *f x y;* . . . . . . *a* . . . . *b* . . . . **end**
write
function *f x y a b;* . . . . . . *a* . . . . *b* . . . . **end**; *f*(% *a*, *b* %) —> *f;*

Similarly for any number of non-local variables.

It is important to take the precaution of 'freezing in' the non-local variables of any function if there is any danger of the function being used in a context where these variables are no longer available. If you fail to do so the values taken may be those for some quite different variables which happen to correspond to the same identifiers. Here is another example:

```
function apply1ton n f; vars i; 1 —> i;
      loop: if i=< n then f(i); i+1 —> i; goto loop close
end;

vars i; 100 —> i;
function g x; pr(x + i)
end;
apply1ton(3, g);
```

The function *apply1ton(n, f)* is intended to execute $f(1), f(2), \ldots, f(n)$.

We expect
101 102 103
but we get
2 4 6.

The reason is, of course, that we have used *i* as a non-local variable in *g*, but when *g* is called in *apply1ton* the *i* referred to will be the most recent one, that is, the local variable *i* of *apply1ton*. Instead of **function** *g* . . . **end;** , we should have written **vars** *g*; **lambda** *x i*; *pr*(*x*+*i*) **end** (% *i* %) —> *g*; .

Another, easier, way of avoiding this difficulty in many cases is given in section 21 'Cancel and sections'.

Of course, sometimes we might intend a non-local to refer to the most recent variable having that identifier, for example, if we had written some functions and wanted to check them out by calling a function *peep* every now and then to print out the values of selected variables:

```
function peep; if trace then [%"peep", i, j, k %] => close end;
```

Similarly if we have a longish program with several functions calling each other, the inner ones having as non-local variables some variables which are local to functions further out, we can adopt the strategy of nesting the function definitions inside each other thus

```
function f x; vars i;
    function g y; ......i....
    end;
...g(x+1)...
end
```

Alternatively, since the non-local *i* will always refer to the most recently occurring *i*, we can write

```
function g y; ......i...
end;
function f x; vars i;
    ....g(x+1)...
end;
```

In the first case we cannot test *g* independently of *f*, because *f* is a local to *g* and cannot be called outside; in the second we can, provided we declare *i* and give it a value.

```
vars i; 99 -> i;
g(1),g(2),g(3) =>
```

Readers familiar with ALGOL 60 will see that the POP-2 way of handling non-local variables is in some ways less convenient than the ALGOL one because, as in the *apply1ton* example, it can lead to mistakes if the proper precautions are neglected. On the other hand, it gives greater flexibility and allows one to write programs less deeply nested, which is particularly useful for on-line debugging. It also allows functions to be produced as the results of other functions, which is quite impracticable with the ALGOL 60 way of handling non-locals. This adds greatly to the power of the language. The idea of a function which had attached to it the values of its non-local variables was suggested by Landin (1964) who called such a function a *closure*.

As another example consider the definition of a function called, say, *twice*, which takes as parameter any function and produces as result the same function applied twice. Thus *twice*(*sqrt*) is a function which computes the square root of the square root of a number, that is, the fourth root.

The obvious but incorrect definition of *twice* is

```
function twice f;
lambda x; f(f(x)) end
end;
```

This will not work because the value of *f* is local to *twice* and will hence be available only during the execution of *twice*. Partial

application enables us to 'freeze in' the value of *f* into the definition of the lambda expression. To do this, the definition of *twice* becomes

```
function twice f;
lambda x f; f(f(x)) end (% f %)
end;

vars root4;
twice(sqrt) —> root4; root4(16) =>
**2.00
function add1 x; x+1 end;
twice(add1) —> add2; add2(5) =>
**7
```

We have now used *twice* to produce two different functions. They are quite independent, and the first one still works correctly.

```
root4(16) =>
**2.00
```

We cannot write *twice(add1)(5)* but we may write *(twice(add1))(5)* or *apply (5, twice(add1))*, using **function** *apply xf; f(x)* **end.**

If we want to examine the values of any parameters which have been frozen into a function by partial application we can do so using the standard function *frozval*, for example,

```
f(% 8, 9 %) —> f1; frozval(1, f1) =>
**8
7 —> frozval(1, f1); frozval(1, f1) =>
**7
```

If we want to do the same thing inside a function we can do it most easily by making the frozen formal parameter into a *reference*. References are a standard class of records with only one component; compare pairs which have two, they are constructed by *consref* and accessed by *cont*. For example,

```
function counter r n; cont(r)+1 —> cont(r); cont(r)=< n end;
counter(% consref(0), 3 %) —> c;
.c, .c, .c, .c, .c =>
** true, true, true, false, false
```

Again the reader acquainted with ALGOL will recognize the analogy with various concepts of **own** variable. We will see in the next section that *c* is an example of a kind of function called a *repeater*, useful for representing such things as input files or streams.

EXERCISES

1. Define a function *sqrts* to find the square roots of a list of functions, e.g. *sqrts*([1 2 3]) = [1.00 1.41 1.73]. (Use *maplist* and partial application.)

2. If *member* (*x*, *s*) is true just if *x* is a member of the set *s* (represented by a list), create a function of one argument which tests whether the argument is a member of the set [2 3 5 7 11 13 17 19].

3. What is the effect of the following program?

```
vars a;
function f x; x —> a end;
8 —> a;
function gx; vars a; 88 —> a; f(x+90); a => end;
g(9);
a =>
```

4. (a) Write a function **, an operation of precedence 3 such that $f^{**}g$ is a function h with $h(x) = g(f(x))$. $f^{**}g$ is usually called the composition of f and g.
(b) What is the value of *maplist*([% *s*, *y*, *z* %], *cos****sin*)?
(c) Define & as an operation of precedence 5 and assign *apply* to it. What is the value of [*1 2 3*] & *twice*(*tl*)***hd*?
(d) What does the function *hd*****nonop** = (% "*monkey*" %) do?
(e) If p is a predicate, i.e., $p(x)$ is a truth value, what is p***not*?

5. Define a function *maketimebomb* such that *maketimebomb*(n) is a function of no arguments, which has no effect the first n—*1* times it is called and prints "*explode*" the nth time. Write a function called *defuse* which renders such a timebomb harmless.

6. Write a version of *maplist* whose result is a dynamic list.

20. INPUT AND OUTPUT FACILITIES

Some of the basic input and output facilities for the console have already been mentioned in previous sections. There are, however, further facilities for dealing with devices other than the on-line console. The whole range of input and output facilities are described in this section. Output facilities are described first because they are somewhat simpler.

OUTPUT FACILITIES

Standard output functions are:

=>	to print the entire contents of the stack on a new line from bottom to top leaving it empty. (Only the top value on the stack is printed if => is used within a function body.)
pr(x)	to print the value of x.
print(x)	to print the value of x and also leave it unchanged on the stack as a result.
nl(n)	to cause further output to continue on a new line leaving n—1 blank lines.
sp(n)	to skip n spaces across the page.
charout(n)	to output the character whose internal code is n. The character code is given as an appendix to the reference manual.
prreal(x, *n1*, *n2*)	to print a real quantity x in a format with n1 places before the decimal point and n2 after.
prstring(x)	prints a string without printing the quotes.

Using these standard output functions, the POP-2 programmer can print results on the console in a flexible manner. Normally programs and data will be kept in some filing system, for example, on a disc store, and the user should consult the description of the filing system for his installation (see, for example, the '*EASYFILE*' system described in Part 4 'Program Library'). The remainder of this section describes the basic facilities for handling devices. Many users will not need to know these details, relying instead on the filing system which will itself make use of these facilities.

INPUT FACILITIES

The following input functions are standard:

itemread ()	to read one word, number or symbol, such as comma or colon.
charin ()	to read one character.

The functions *charin* and *itemread* are the normal way a POP-2 program reads its data from the console keyboard. For example, a Program to read a set of integers and print their total when the word "*end*" is typed could be defined as follows:

```
function sum;
vars x total;k0: 0 -> total;
k1: itemread() -> x;
if x = "end" then pr(total);goto k0 close;
total + x -> total; goto k1
end
```

When the function *sum* is executed, it will read items typed on the keyboard. When an integer is typed it is added into the total. When the word *end* is typed the total is printed and the process starts over again. If anything other than an integer or the word *end* is typed, attempting to add it to total will result in an error message, and further console input will be POP-2 text rather than data read by *sum*. This is one way of terminating the otherwise infinite program. The other way is to hit the key on the console which interrupts the POP-2 program and then type *setpop*(), which returns the system in readiness for program input.

REPEATERS AND CONSUMERS

There are two kinds of function, which must be introduced, to explain the input/output mechanism of POP-2. We call them a *repeater* (for input) and a *consumer* (for output). They are complementary in the way that select and update functions are complementary, and indeed they can be regarded as special kinds of select and update functions respectively.

To illustrate the idea in a familiar context consider the following definitions:

```
vars inlist outlist;
function fromlist; vars x; inlist.hd -> x; inlist.tl -> inlist; x end;
function tolist x; outlist <> [% x %] -> outlist end;
```

Thus *fromlist*() produces the next item on the *inlist*, and *tolist*(*x*) appends *x* to the end of *outlist*.

```
[1 2 3 4] -> inlist; nil -> outlist;
   tolist(2*fromlist());
   tolist(2*fromlist());
   tolist(2*fromlist());
   outlist =>
** [2 4 6]
```

Compare this with

```
pr(2*itemread());
pr(2*itemread());
pr(2*itemread());
```

If the input file is 1, 2, 3, the output file will be 2, 4, 6. The situations are strictly analogous. *fromlist and itemread* are repeaters (they repeatedly produce an item), whilst *tolist* and *pr* are consumers (they consume items).

To show how they can be viewed as select-update doublets we can say

```
vars list;fromlist -> list;tolist -> updater(list);
[1 2 3]-> inlist;nil -> outlist;
2*list() -> list();
2*list() -> list();
2*list() -> list();
outlist =>
**[2 4 6]
```

Note. If we want to do the same with *itemread* and *pr* a slight difficulty arises.

```
vars console;itemread -> console;pr -> updater(console);
```

would be wrong because the value of *itemread* is a standard function and the system protects it and will not allow its updater to be altered. However, if we partially apply *no* arguments into *itemread* we get a new function whose updater can be assigned to.

```
vars console;itemread(%%) -> console;pr -> updater(console);
console() -> console():
1       (input)
1       (output)
2       (input)
2       (output)
```

CHANGING THE INPUT OR OUTPUT DEVICE

There are several kinds of repeaters and consumers for input and output. First they may produce either a character or an item, such as a word or unsigned number. Secondly the source or destination may vary, for example, it might be the console, or paper tape, or disc.

One source, normally the console, and one destination, normally the console, are taken as standard for any implementation of POP-2. When the user starts to use the system it compiles his program and reads data from the standard input device. He may later cause program to be compiled or data to be read from other devices, but there must be some means of communication specified a priori. Likewise his results come out on the standard output device until he decides to use some other device.

The standard variable *cucharout* has, as value, a consumer for output of characters which enables the programmer to define his own output routines for nonstandard output. In fact all the standard output functions such as *pr* and *nl* use the variable *cucharout.* It normally has as its value the standard function *charout* for output to the console. All that is necessary, therefore, to cause a program to output its results to some device other than the console is to replace the current value of *cucharout* with an equivalent function for the device required. The assignment "*charout -> cucharout*" can be used to restore output to the console. This is done automatically if an error occurs.

There is a standard function *popmess* (short for 'pop message') which produces as result character repeater and/or consumer functions for devices other than the console. To cause further output to be printed on a line printer with the heading, say,

[*xyz program results*]

we have only to execute the assignment

```
popmess([lp80 xyz program results]) -> cucharout;
```

assuming *lp80* is the appropriate device name for the line printer. The available list of device names and the layout of the arguments of *popmess* may differ from one installation to another. Because devices other than the console may be shared among several users, *popmess* must first check that the line printer is not already in use before returning with the character output function for the line printer. The device is returned to the pool when the output file is closed by outputting the item *termin*. This can be done by executing the statement

pr(termin)

After this the device is again available for other users, and any attempt to continue outputting to the line printer will result in an error.

To restore output to the console, it is not necessary to call *popmess*, because the console is permanently allocated to the user. The character output function for the console is *charout* and it is only necessary to execute the assignment

charout -> cucharout

after which further output appears on the console.

Now consider how *sum*, defined earlier to read items and add them up, might have been defined to process information from any input such as paper tape or cards. Clearly the input device must be made a parameter of *sum*. The definition might be written thus:

```
function sum i;
vars x total; k0: 0 -> total;
k1: i() -> x;
if x = "end" then pr(total); goto k0 close
total + x total; goto k1
end
```

Now when *sum* is called it must be given as a parameter a repeater function to read items from the chosen device. To read from the keyboard just like the first version, it is called by executing the statement

sum (itemread)

For any other device, a function corresponding to itemread is needed. Just as *popmess* gets character output functions for output devices, it also gets character input functions for input devices. To obtain a character input function for a paper tape

[abc data]

the following must be executed

popmess ([ptin abc data]) -> x

which makes *x* a character input function. Successive calls of *x* produce the successive characters of the paper tape file. *x* is not, however, a suitable argument for *sum*, which needs an item repeater, that is, item-producing function, rather than a character repeater.

There is a standard function *incharitem* which takes a character repeater and produces as result an item repeater. Thus the value of

incharitem (x)

is a function just like *itemread*, but which reads its data from a source other than the console keyboard. In fact *itemread* could be defined as *incharitem (charin)* except that, as is explained below, *itemread* always reads from the current source of POP-2 text and not just from the console.

Having thus opened a paper tape file for reading, the function *sum* can be called to process it by executing the statement

```
sum(incharitem(x))
```

after which the tape will be read and whenever the word "*end*." is encountered, the accumulated total will be printed on the console. Attempting to process beyond the end of the file will cause an error message which terminates execution of *sum*.

EXECUTING POP-2 TEXT FROM DEVICES OTHER THAN THE CONSOLE.

The POP-2 system normally reads and executes POP-2 text from the console keyboard. It is convenient to be able to input and execute text from faster devices in order to input established function definitions and data structures. Some knowledge of the mechanism involved in reading POP-2 text helps in the understanding of how other devices can be used.

The standard function *compile* takes a character repeater as argument and executes the sequence of characters as POP-2 text. Using *compile*, all that is necessary to execute POP-2 text punched on a paper tape file called, say, [*xyz prog*] would be the two statements

```
popmess([ptin xyz prog] —> x;
compile(x);
```

The POP-2 system takes its input from a list called *proglist*. Because items are normally typed in at the console, *proglist* is normally defined to be a dynamic list in which the rule for getting the next item is actually an item-producing function such as *incharitem* (*charin*). A simple demonstration of this mechanism can be obtained by typing

```
[2 + 2 =>] <> proglist —> proglist;
```

which joins the list of four items [2 + 2 =>] onto the beginning of *proglist*. When the assignment has been executed, the system returns to getting its input from *proglist*, thus executing the statement

```
2 + 2 =>
```

and printing the result

```
**4
```

on the console.

All that is necessary to make the system accept POP-2 text from any source is to turn the source into a list (probably a dynamic list) and join it onto the beginning of *proglist*. It was shown that *incharitem* (*x*), where *x* is a character input function obtained from *popmess*, is an item-reading function. The standard function *fntolist* can therefore be used to turn it into a list ready for joining onto *proglist*. The standard function *compile* which facilitates this could be defined as follows:

```
function compile x;
fntolist(incharitem(x)) <> proglist —> proglist
end
```

The function *itemread* actually removes items from the head of *proglist*. It follows that while a program tape is being compiled, execution of *itemread* causes items to be read from the paper tape file. The paper tape file can therefore consist of exactly the same information as would by typed directly on the keyboard.

The input mechanism described above may appear rather complex. It does, however, provide the user with access to information being processed by the POP-2 system. For example, it would be very easy to execute a POP-2 program on paper tape in which occurrences of the word *function* had been abbreviated by the word *fn*. The following would suffice:

```
popmess([ptin xyz prog]) -> x;
function edit i;
vars k; i() -> k;
if k = "fn" then "function" else k close
end;
fntolist(edit(%_incharitem(x)_%)) <> proglist -> proglist;
```

Having opened the paper tape file and assigned the appropriate character input function to *x*, an auxiliary function *edit* could be defined. The function *edit* takes an item-reading function and produces, as result, an item. By partially applying *edit* to the item-reading function for the appropriate input device, a function results which, when called successively, yields the successive items of the file, with editing where appropriate. This is then in a form which can be turned into a dynamic list by *fntolist* and joined to the start of proglist. A somewhat different way of editing input is given in the program *'POPEDIT'* described in the Program Library (see Part 4). It edits character repeaters rather than item repeaters.

EXERCISES

1. Write a function to print a neat table of *sqrt(x*x+y*y)* for *x* and *y* from 0 to 8 in steps of 1 with 3 places of decimals.

2. Write an integer repeater to produce the even integers 0, 2, 4, 6, 8, . . .

3. Write a function which takes an integer repeater and produces a real repeater, giving the square roots of the integers.

Write one which produces an integer repeater being the sums of successive pairs of integers.

4. Write a function *printprog* to enable one to print out a program on any device while it is being compiled. Thus typing

compile(printprog(r, c))

reads text using the character repeater *r* for input and also prints it on an output device which has the character consumer *c*.

21. CANCEL AND SECTIONS

There are times when we wish to get rid of an identifier, perhaps because we wish to use the same identifier for some other purpose. We may do this by writing

cancel *x*;

We must distinguish between the identifier *x* which is cancelled and the variable associated with this identifier, that is, the actual pigeonhole in the machine used to hold the value. This pigeonhole is not destroyed and indeed any functions already compiled which refer to it go on doing so, but we may no longer refer to it by including the symbol *x* in our program. If we declare *x* again by **vars** *x*; a new pigeonhole (variable) will be created quite separate from the old one.

Thus one use of cancelling is to ensure that functions already compiled which use *x* cannot be interfered with by using *x* for any other purpose. For example,

```
function sigma f n => s; vars i; 0 -> s; 0 -> i;
    loop: if i=<n then f(i) + s -> s; i + 1 -> i; goto loop;
end;
cancel f i n s;
vars n; 3 -> n;
sigma(lambda x; x↑n end, 10) =>
```

gives the expected sum of cubes but would have gone wrong (giving a sum of tenth powers) if we had not cancelled *n*. As explained in the section on partial application, instead of cancelling *n* we could have bound in the value of *n* by writing

```
sigma(lambda x↑n; x n end (% n %), 10) =>
```

It may be that we just want to get rid of such an identifier (or, more precisely, break its association with certain variables) temporarily and revive it later. We can do this by writing part of the program as a *section*.

Any identifiers declared in this section of the program then have no connection with those used outside it, as if their names had all been systematically changed so as to be distinct from identifiers outside. Thus

```
vars x y; 1 -> x; 2 -> y;
section;
    vars x y; 100 -> x; 200 -> y
        x + y =>
**300
endsection;
x + y =>
**3
```

just as if it had been written

```
section;
    vars x999 y999; 100 -> x999; 200 -> y999;
        x999 + y999 =>
    **300
endsection;
```

A section may have a name, and we could have written

```
section addition;
```

Of course we may wish to use some functions or other data defined in the section later on outside, that is, we might want to declare some identifiers inside the section and then have them usable afterwards outside. Such identifiers are called 'external identifiers'. Here is an example

```
function f; ....... end;
vars x y; 1 -> x; 2 -> y;
section first => g; vars y; 3 -> y;
    function h; ....... end;
    function g z; ...x...y...f...h...end
endsection;
g(x+y) =>
```

g is an external identifier declared in the section named *first* for use afterwards outside. Note that the function *h* defined in the section could not be used outside, nor could the *y* in the section have any connection with the *y* outside, but the *x* and the *f* used in *g* are the same as the ones outside since they have not been redeclared. We can get rid of the identifiers produced by a section such as *first* by just typing **cancel** *first*.

If one is writing functions for inclusion in a program library so that they may be incorporated in other people's programs, it is wise to enclose them in a section so as to avoid any unintentional clash of identifiers.

Sections can also be used to solve the problem associated with using a function with non-local variables as a parameter of another function (this problem was discussed in section 19 'Partial application'). We simply make the definition of the function which has a function parameter into a separate section. Then its local identifiers cannot clash accidentally with those used outside. Using the same example as before, we write

```
section => apply1ton;
    function apply1ton n f; vars i; 1 -> i;
        loop: if i =< n then f(i);i+1 -> i;goto loop close
    end;
endsection;
vars i; 100 -> i;
function g x; pr(x+1)
end;
apply1ton(3,g);
101 102 103
```

The global *i* which receives the value 100 is now quite distinct from the *i* declared in *apply1ton* and we get the desired results.

22. MACROS AND POPVAL

Sometimes a particular piece of program has to be written over and over again with only minor changes. Usually one can define a function to effect this piece of program, making the changeable parts parameters. Sometimes this is undesirable, for example, if the time taken to enter the function and exit from it would slow the program down considerably, or it is impossible, for example, because the changeable parts are not suitable for making into function parameters. For example,

```
if dataword(x) = "complex" then
```

might occur frequently and although we could define

```
function dcomplex x; dataword(x) = "complex" end
```

speed might be too important to allow this. An example where it is impossible to define a function would be if the following statement occurred frequently:

```
0 -> i;
loop: if i = n then ....; i + 1 -> i;goto loop close
```

We cannot make the statements represented by .... into a parameter (unless, clumsily, we make them a lambda expression with no arguments).

This difficulty can be overcome by defining a macro, a means of generating a piece of POP-2 text during compilation, possibly with some variations. A simple example with no variation.

```
macro zeroxyz; macresults([0 -> x ;0 -> y ;0 -> z; ]) end;
```

From here on, whenever the identifier *zeroxyz* appears in the program it is as if *0 -> x; 0 -> y ;0 -> z;* had been written instead. Note that the text is enclosed in list brackets and made the argument of the standard function *macresults.*

In fact a macro is just a function with the curious property that it is executed during compilation. To get some variety we may make the macro read the word or words which follow it and, for example, place them somewhere in the output list.

```
macro initxyz; vars a;  .itemread -> a
    macresults([% a, "-> ", "x", ";", a, "-> ", "y", ";",
                  ", a, "->", "z", ";" %])
end;
initxyz 3;
x, y, z =>
**3, 3, 3
```

Our first example above could be handled by

```
macro dcomplex; vars x;  .itemread -> x;
    macresults([dataword ([] <> [% x %] <> []) = "complex"])
end;
....
    if dcomplex y then ..
```

Note the limitations of macros, this one would not enable us to write

```
if dcomplex hd(y) then
```

The reader may like to define a macro **cycle** and a macro **repeat** so that we can write

```
cycle i = n;
  ....
repeat i
```

instead of

```
0 -> i;
loop: if i < n then ... ;i + 1 -> i;goto loop close
```

or, better, instead of

```
loopif i < n then ...; i + 1 -> i close
```

Just as one may occasionally want to execute program at compile time, using a macro, one may occasionally want to compile program at execute time. A standard function *popval* is provided for this purpose. It takes a piece of program in the form of a list, and compiles and executes it. The list should have the special word **goon** (go on) as its last item and when this is reached *popval* exits and the computation continues normally. The items in the list are words, numbers, and strings, for example,

```
popval([x + y => goon]);
popval(['so far so good' -> status;goon]);
popval([function f x;x*x end;f(9) => goon]);
```

Although *popval* can be used inside a function the program text is to be thought of as if it had occurred at the outer level of the program, not in the body of any function (but still in the current section). Any

variables mentioned take their most recent values however, so that in the first example *x* and *y* might refer to local variables of the function in which *popval* is applied.

Naturally if we know in advance the piece of program to which *popval* is to be applied we might as well just write in that piece of program, so that *popval* is most useful when this is not known until execute time, for example, a program which asks its user to type in any arithmetic expression as data and then compiles it as a function and numerically integrates it for him.

If the operating system of the particular implementation allows it, a means of interrupting program execution may be provided, for example, by depressing a special key on the console. Any text typed in up to the word **goon** will then be executed, just as if it had been the argument of a *popval* statement inserted at that point in the program. This enables us to examine the state of a program and perhaps change the values of some variables, and then let it continue.

EXERCISES

1. (a) Write a macro *p1function* so that writing

```
p1function f x;
```

for any *f* and *x* is equivalent to writing

```
function f x;x =>
```

so long as a variable *pfun* is set to true, otherwise equivalent to **function** *f x;*
(You could use this as an aid to finding mistakes in a program.)

(b) Elaborate *p1function* to deal with functions with any number of arguments. Call it *pfunction.*

2. Write a macro —>> so that
e —>> *x*, *y*, *z*;
is equivalent to
e —> *x*; *x* —> *y*; *y* —> *z*;

3. Use *popval* to write a function to read in an arithmetic expression in the variable *x* from the console and print its values for integers *x* between 1 and 10.

23. JUMPOUT

The following piece of POP-2 program is *illegal.*

```
function f x;
    if x=0 then goto error close;
    (x + 1)/x
end;
function g y;
    f(y) + f(y↑3) =>
    goto last;
error: 'zero error' =>
last:
end
```

This mistake is that a **goto** statement cannot refer to a label outside the function body in which it occurs. In this case we can obtain the

desired effect by using a special standard function *jumpout*. We write, for example,

```
vars error;
function f x;
    if x=0 then error() close;
    (x + 1)/x
end;

function g y; jumpout(lambda; 'zero error' => end, 0) -> error;
    f(y) + f(y↑3) =>
end;
```

Thus instead of a label *error* we have a function *error* of no arguments and no results produced by *jumpout* from the function **lambda;** *'zero error'* => **end.** In fact, *error* is identical to this latter function except that, as soon as it has been executed, execution of *g* is terminated instead of execution of *f* being resumed as one would normally expect. Thus instead of the normal exit mechanism *error* has a special 'fire-escape' which enables it to climb out of *g* when it is called (*g* is the function where *error* was created by *jumpout*).

The second parameter, 0, of *jumpout*, indicates that the function produces no results. A case where *jumpout* would be applied to a function with a result would be the successful conclusion of a search process. For example, given a binary tree represented by a list structure with numbers at the tips we might want to find some number greater than *n* on it.

```
function search t n; vars answer;
    jumpout(lambda x ;x end, 1) -> answer;
    function test t;
        if not(atom(t)) then test(t. hd); test(t. tl) close;
        if t > n then answer(t) close
    end;
    test(t); undef
end;
vars tree;
(1::6)::(1::4) -> tree;
search(tree, 3) =>
**6
search(tree, 10) =>
**undef
```

The note on page 279 describes a more general jump facility.

24. SOME USEFUL STANDARD FUNCTIONS

There are a few facilities which logically would have been introduced in earlier chapters but were omitted in order not to burden the description with too much detail.

BIT MANIPULATION

One sometimes wants to perform operations on patterns of bits (binary digits 0 or 1) such as taking the logical *and* of two patterns, for example, logical and (0011, 0110) = 0010, or the logical *or*, for example, logical or (0011, 0110) = 0111.

Integers may be regarded as such bit-strings, the number of binary

digits allowed depending on the largest integer allowed by the implementation. Standard functions *logand, logor,* and *lognot* are provided. Thus

logand(15, lognot(logand(3, 6))) =>
**13

The integers may also be written in binary or octal by prefixing 2: or 8:, thus

2:0011 =>
**3
8: 77 =>
**63

The standard function *logshift* allows shifting the pattern to the left direction by plus or minus *n* binary places

logshift(2:0011, 3) =>
**24
logshift(8:77, —2) =>
**15

BOOLEAN FUNCTIONS

The symbols **and** and **or** used in conditional expressions are not functions because they do not evaluate both their arguments. They are better regarded as abbreviations for certain kinds of conditional expressions. The corresponding functions are provided, as standard, called *booland* and *boolor*. Thus, for example,

booland(true, boolor(false, true))
has value *true*.

FNPROPS, MEANING AND IDENTPROPS

It is sometimes useful to attach some arbitrary piece of information to a function or a word, for example, one might attach to a function the number of parameters it requires. Standard doublets *fnprops* and *meaning* are provided for this purpose.

3 —> fnprops (f);
fnprops (f) =>
**3
"noun" —> meaning("house"); "verb" —> meaning("lives");

There is also a standard function to find properties of an identifier or syntax word, for example

vars operation *6 i;*
identprops ("l") =>
**6
identprops("then")
***syntax*

DATALENGTH AND DATALIST

Standard functions are provided to find the number of components in a strip or record and to produce a list of these components. Thus

vars *characters; initc(10) —> characters;*
datalength(characters) =>
**10

If *consper* constructs a record

datalist(*consper*("*smith*", *31*, *0*)) =>
**[*smith 31 0*]

APPENDIX TO PRIMER

ANSWERS TO EXERCISES

Note. The function *next* appears occasionally instead of *dest*. This is a mistake since *next* is not a standard function in revised POP-2 (it was previously a synonym for *dest*).

```
14.19HRS.  14 MAR 1970.                                          ** DBA **

[DBA BIGSHOOT]

      COMMENT
      THIS IS THE FIRST OF A SET OF FILES OF POP-2 TEXT WHICH ARE
      THE ANSWERS TO THE EXAMPLES IN THE PRIMER.  ALL THE FILES
      CAN BE COMPILED: WHERE THE QUESTION ASKS YOU TO WRITE A
      FUNCTION THIS FUNCTION WILL THEN BE READY TO USE: OTHER
      ANSWERS ARE GIVEN AS COMMENTS.
      THE ANSWERS ARE, OF COURSE, NOT UNIQUE: WE HAVE TRIED TO
      MAKE THEM STRAIGHTFORWARD RATHER THAN ELEGANT OR EFFICIENT.
      THEY HAVE ALL BEEN TESTED ON THE POP-2 SYSTEM AT MACHINE
      INTELLIGENCE EDINBURGH, BUT NOTIFICATION OF ANY ERRORS OR
      OMISSIONS WOULD BE WELCOME.
      TAPES OF THESE FILES WILL BE MADE AVAILABLE WITH THE POP-2
      SOFTWARE SYSTEM.
                              BRUCE ANDERSON  EDINBURGH JULY 1969 ;
      COMMENT
      NO EXERCISES IN SECTION 1;


      COMMENT
2.1   (A)  (2.5*2)/(-1.5*4)=>
      (B)  1+2*(5-3)=>
      (C)  SQRT(3↑2 + 4↑2)=>
      (D)  (SIN(0.13))↑2 + (COS(0.13))↑2 =>
      (E)  ARCTAN(1.5)=>

2.2   (A)  24.0
      (B)  1.16
      (C)  42.0

2.3   (A)  ) MISSING
      (B)  SHOULD BE () AROUND THE 0.5
      (C)  THE EXPONENT MUST BE AN AN INTEGER
      (D)  6. IS NOT ALLOWED
      ;
      COMMENT

3.1   VARS Z; X->Z; Y->X; Z->Y;

3.2   (A)  **11
      (B)  **6,78,13

3.3   VARS TRIG;
      SQRT(SIN(X+A))->TRIG;
      TRIG*TRIG=>
      ';
      COMMENT

4.1   14

4.2   (A) SWAPS THE VALUES OF X AND Y
      (B) SWAPS THE VALUES OF THE TOP TWO ITEMS ON THE
         STACK, THOUGH IF THERE ARE LESS THAN TWO ITEMS ON THE STACK AN
         ERROR WILL RESULT - IT IS CALLED STACK UNDERFLOW

4.3   VARS A B C;
      ->A ->B ->C;
       A,B,C;
       ;
      COMMENT

5.1   HERE ARE TWO FUNCTIONS TO DO QUADRATIC EQUATIONS, THE SECOND
      ONE AVOIDING DOING THE SAME CALCULATION TWICE ;

      FUNCTION ROOTS A B C;
        (-B + SQRT(B↑2 - 4*A*C))/(2*A),
        (-B - SQRT(B↑2 - 4*A*C))/(2*A)
      END;
```

```
     FUNCTION ROOTS2 A B C;
       VARS U V;
       -B/(2*A)->U;
       (SQRT(B↑2 - 4*A*C))/(2*A) ->V;
       U+V,U-V
     END;


     COMMENT
5.2  **144


5.3  TYPE  APPLY1TON(31,PRNEWEXPR) AFTER DEFINING THE FOLLOWING
     FUNCTIONS.;

     FUNCTION EXPR X;
       1 + X + 0.5*X↑2 + (1/6)*X↑3
     END;

     FUNCTION PRNEWEXPR Y;
       EXPR((Y-1)/10)=>
     END;


     COMMENT
5.1; FUNCTION ROOTS3 A B C;
       VARS U V;
       IF A=0 THEN -C/B,0,-C/B,0 EXIT;
       B↑2 - 4*A*C ->V; -B/(2*A)->U;
       IF V>=0 THEN SQRT(V)/(2*A)->V; U+V,0,U-V,0
               ELSE SQRT(-V)/(2*A)->V; U,V,U,-V
       CLOSE
     END;


     COMMENT
5.2; FUNCTION ISOK N;
       IF N<100 AND ERASE(N//3)=0 OR ERASE(N//4)=0 OR ERASE( N//5)=0
         THEN TRUE
         ELSE FALSE
       CLOSE
     END;


     COMMENT
5.3; FUNCTION TAX I;
           IF I =< 150 THEN 0
       ELSEIF I =< 400 THEN (I-150)/10
       ELSEIF I =< 600 THEN 25 + (I-400)/4
                       ELSE 75 + (I-600)/3
           CLOSE
     END;


     COMMENT
5.4; FUNCTION ORDER3 X Y Z;
           IF X>Y THEN Y,X->Y ->X CLOSE;
           IF Z<X THEN Z,X,Y
       ELSEIF Z>Y THEN X,Y,Z
                  ELSE X,Z,Y
           CLOSE
     END;
```

```
    COMMENT
7.1; FUNCTION TAB2 FUN XLO DX XHI YLO DY YHI;
       VARS X Y VALUE;
       XLO->X; YLO->Y;
     COL: IF X>XHI THEN RETURN ELSE NL(1) CLOSE;
     ROW: IF Y>YHI THEN YLO->Y; X+DX->X; GOTO COL CLOSE;
          FUN(X,Y)->VALUE;
          IF VALUE<10 THEN SP(2) ELSEIF VALUE<100 THEN SP(1) CLOSE;
          PR(VALUE); SP(2);
          Y+DY->Y;
          GOTO ROW
     END;

     FUNCTION TIMES X Y; X*Y END;

     COMMENT TO USE THE FUNCTION AS ASKED IN THE QUESTION, TYPE
     TAB2(TIMES,1,1,10,1,1,10);


7.2; FUNCTION ABS X;
       IF X<0 THEN -X ELSE X CLOSE
     END;

     FUNCTION TERMS  N EPSILON;
       VARS K XK SQRTN;
       0->K; 1->XK; SQRT(N)->SQRTN;
     LOOP: IF ABS(SQRTN-XK)=<EPSILON THEN K; RETURN CLOSE;
           0.5*(XK + N/XK)->XK;
           K+1->K;
           GOTO LOOP
     END;


     COMMENT
7.3; FUNCTION APPLY1TON N F;
       VARS INT; 1->INT;
     LOOP: F(INT);
           INT+1->INT;
           IF INT=<N THEN GOTO LOOP CLOSE
     END;


      COMMENT
      NO EXERCISES IN SECTION 8;


     COMMENT
9.1  (A)   OUT(20,"POUNDS");
           20 POUNDS,PLEASE  OUT(40,"DOLLARS");
           40 DOLLARS,PLEASE
     (B)   **TRUE


9.2';FUNCTION FIRSTLETTER WORD;
       VARS N;
       CHARWORD(WORD); ->N;
     LOOP: IF N=1 THEN EXIT;
           ERASE(); N-1->N;
           GOTO LOOP
     END;

     FUNCTION ORDER WORD1 WORD2;
       FIRSTLETTER(WORD1)->WORD1;
       FIRSTLETTER(WORD2)->WORD2;
           IF WORD1>WORD2 THEN "AFTER"
       ELSEIF WORD1=WORD2 THEN "SAME"
                          ELSE "BEFORE"
           CLOSE
     END;
```

```
      COMMENT
10.1: FUNCTION EXISTS XL P;
      LOOP: IF NULL(XL)  THEN FALSE EXIT;
            IF P(HD(XL)) THEN TRUE; RETURN
                         ELSE TL(XL)->XL; GOTO LOOP
            CLOSE
      END;


      COMMENT
10.2: FUNCTION APPEND X XL;
        XL<>(X::NIL)
      END;

      FUNCTION DELETE X XL=>RESULT;
        NIL->RESULT;
      LOOP: IF NULL(XL) THEN EXIT;
            IF NOT(HD(XL)=X) THEN APPEND(HD(XL),RESULT)->RESULT;CLOSE;
            TL(XL)->XL;
            GOTO LOOP
      END;


      COMMENT
10.3: FUNCTION ASSOC X XYL => Y;
        VARS X1;
      LOOP: IF NULL(XYL)  THEN UNDEF->Y
                          ELSE HD(XYL)->X1; TL(XYL)->XYL;
                               IF X1=X THEN HD(XYL)->Y
                                       ELSE TL(XYL)->XYL;
                                            GOTO LOOP
                               CLOSE

            CLOSE
      END;

      FUNCTION MAKEASSOC X Y XYL => XYL1;
        X::(Y::XYL)->XYL1
      END;

      VARS PRICE;
      [FAGS 50 MILK 11 SUGAR 15 BEER 28 CARROTS 40]->PRICE;

      FUNCTION COST LIST => TOTAL;
        0->TOTAL;
      LOOP: IF NULL(LIST) THEN EXIT;
            ASSOC(HD(LIST),PRICE)+TOTAL->TOTAL;
            LIST.TL->LIST;
            GOTO LOOP
      END;


      COMMENT
10.4: FUNCTION SAME XL1 XL2;
      LOOP: IF NULL(XL1) OR NULL(XL2) THEN XL1=XL2; RETURN CLOSE;
            IF HD(XL1)=HD(XL2) THEN TL(XL1)->XL1;
                                    TL(XL2)->XL2; GOTO LOOP
                               ELSE FALSE
            CLOSE
      END;

      FUNCTION WANTED CRIMLIST DESCRIP;
        VARS SUSPECTS THISONE; NIL->SUSPECTS;
      LOOP: IF NULL(CRIMLIST) THEN SUSPECTS EXIT;
            HD(CRIMLIST)->THISONE; TL(CRIMLIST)->CRIMLIST;
            IF SAME(TL(TL(THISONE)),DESCRIP)
              THEN (HD(TL(THISONE)))::SUSPECTS->SUSPECTS
            CLOSE
            GOTO LOOP
      END;
```

```
VARS CRIMINALS;
[[NAME JONES HAIR SANDY EYES BROWN HEIGHT 65]
 [NAME CRIPPEN HAIR NONE EYES GREEN HEIGHT 61]
 [NAME POP HAIR VERY EYES TWO HEIGHT 69]
 [NAME DIZZY HAIR SANDY EYES BLUE HEIGHT 66]
 [NAME BERT HAIR NONE EYES GREEN HEIGHT 61]]->CRIMINALS;


      COMMENT
10.5; FUNCTION MEMBER XX XXL;
      LOOP:      IF XXL.NULL  THEN FALSE
            ELSEIF XX=XXL.HD THEN TRUE
                             ELSE XXL.TL->XXL; GOTO LOOP
                  CLOSE
      END;

      FUNCTION BIGUNION XLL;
        VARS ANSWER; NIL->ANSWER;
      LOOP: IF XLL.NULL THEN ANSWER EXIT;
            ANSWER<>XLL.HD->ANSWER;
            XLL.TL->XLL;
            GOTO LOOP
      END;

      FUNCTION PRUNE XXL;
        VARS XXL2; NIL->XXL2;
      LOOP:      IF XXL.NULL THEN XXL2; RETURN
            ELSEIF NOT(MEMBER(XXL.HD,XXL.TL))
                 THEN(XXL.HD)::XXL2->XXL2;
                CLOSE;
            XXL.TL->XXL;
            GOTO LOOP
      END;

      VARS FLIGHTLIST;
      [EDIN [LIV] LIV [LOND MANCH] MANCH [LOND EDIN]
      LOND [BRIST TRURO] BRIST [TRURO] TRURO [PARIS LIV]
      PARIS [LOND]]->FLIGHTLIST;

      FUNCTION ASSOCFLIGHTS X;
        ASSOC(X,FLIGHTLIST)
      END;

      FUNCTION REACHABLE PLACE CHANGES;
        VARS CURRENT;
        PLACE::NIL->CURRENT;
      LOOP: PRUNE((BIGUNION(MAPLIST(CURRENT,ASSOCFLIGHTS)))<>CURRENT)
                              ->CURRENT;
            IF CHANGES= 0 THEN CURRENT
                          ELSE CHANGES-1->CHANGES;
                               GOTO LOOP
            CLOSE
      END;

      COMMENT
11.1  TAB(LAMBDA X; X*X - 2*X - 1 END,0,1,100);

11.2  4
       ;

      COMMENT
12.1; FUNCTION HCF N1 N2;
            IF N1<N2 THEN HCF(N2,N1)
        ELSEIF ERASE(N1//N2)=0 THEN N2
                               ELSE HCF(N2,ERASE(N1//N2))
            CLOSE
      END;
```

```
FUNCTION HCF2 N1 N2;
  VARS REM;
  IF N1<N2 THEN N1,N2 ->N1->N2; CLOSE;
LOOP: ERASE(N1//N2)->REM;
      IF REM=0 THEN N2 ELSE N2->N1; REM->N2; GOTO LOOP CLOSE;
END;


      COMMENT
12.2; FUNCTION MAPLIST1 LIST FUN;
        IF LIST.NULL THEN NIL
                     ELSE FUN(LIST.HD)::MAPLIST1(LIST.TL,FUN)
        CLOSE
      END;


      COMMENT
12.3  **10,[4 3 2 1]


12.4; FUNCTION EXISTS XL P;
            IF NULL(XL)  THEN FALSE
        ELSEIF P(HD(XL)) THEN TRUE
                         ELSE EXISTS(TL(XL),P)
            CLOSE
      END;

      FUNCTION DELETE X XL;
            IF XL.NULL THEN NIL
        ELSEIF XL.HD=X THEN DELETE(X,XL.TL)
                       ELSE (XL.HD)::DELETE(X,XL.TL)
            CLOSE
      END;


      COMMENT
13.1; VARS OPERATION 7 /= ;
      LAMBDA LEFT RIGHT; NOT(LEFT=RIGHT) END->NONOP /= ;

      COMMENT
14.1; FUNCTION MAKEASSOC X Y XYL => XYL1;
        VARS XYLP; XYL->XYLP;
      LOOP:       IF XYLP.NULL THEN X::(Y::XYL)->XYL1;
            ELSEIF XYLP.HD=X THEN Y->XYLP.TL.HD; XYL->XYL1;
                             ELSE XYLP.TL.TL->XYLP; GOTO LOOP
                  CLOSE
      END;


      COMMENT
14.2  **1,2,1,2

14.3; FUNCTION EQFRONT XL1 XL2;
      COMMENT 'TRUE IF XL2=XL1<>XL';
            IF NULL(XL1) THEN XL2,TRUE
        ELSEIF NULL(XL2) THEN FALSE
        ELSEIF HD(XL1)=HD(XL2) THEN EQFRONT(TL(XL1),TL(XL2))
                               ELSE FALSE
            CLOSE;
      END;

      FUNCTION EDIT XL OLD NEW;
        VARS REMAINS;
            IF NULL(XL) THEN NIL
        ELSEIF EQFRONT(OLD,XL) THEN ->REMAINS;
                                    NEW<>EDIT(REMAINS,OLD,NEW)
                               ELSE (XL.HD)::EDIT(XL.TL,OLD,NEW)
            CLOSE
      END;
```

```
      COMMENT
14.4; FUNCTION ISPRIME N;
        VARS DIV; 1->DIV;
      LOOP: DIV+1->DIV;
                IF ERASE(N//DIV)=0 THEN FALSE
            ELSEIF DIV*DIV>N THEN TRUE
                             ELSE GOTO LOOP
                CLOSE
      END;

      VARS LASTNUM LIMIT;

      FUNCTION NEXTPRIME;
      LOOP: LASTNUM+1->LASTNUM;
                IF LASTNUM>LIMIT THEN TERMIN
            ELSEIF LASTNUM.ISPRIME THEN LASTNUM
                                   ELSE GOTO LOOP
                CLOSE
      END;

      FUNCTION MAKEPRLIST N;
        0->LASTNUM;
        N->LIMIT;
        FNTOLIST(NEXTPRIME)
      END;


      COMMENT
15.1; VARS YOF XOF DESTPOINT CONSPOINT V3OF V2OF V1OF DESTTRIANGLE
                                                        CONSTRIANGLE;

      RECORDFNS("POINT",[0 0])->YOF ->XOF ->DESTPOINT ->CONSPOINT;
      RECORDFNS("TRIANGLE",[0 0 0])->V3OF ->V2OF ->V1OF
                       ->DESTTRIANGLE ->CONSTRIANGLE;

      FUNCTION DIST P1 P2;
        SQRT((XOF(P1)-XOF(P2))↑2 + (YOF(P1)-YOF(P2))↑2)
      END;

      FUNCTION ABS X;
        IF X<0 THEN -X ELSE X CLOSE
      END;

      VARS OPERATION 7 == ;
      LAMBDA A B; ABS(2*(A-B)/(A+B)) =<0.001 END ->NONOP == ;

      FUNCTION EQUILATERAL TRIANGLE;
        VARS V1 V2 V3;
        V1OF(TRIANGLE)->V1; V2OF(TRIANGLE)->V2; V3OF(TRIANGLE)->V3;
        BOOLAND(DIST(V1,V2)==DIST(V2,V3),DIST(V1,V2)==DIST(V1,V3))
      END;

      VARS PP1 PP2 PP3 PP4 TR1 TR2;
      CONSPOINT(0,0)->PP1; CONSPOINT(2,0)->PP2;
      CONSPOINT(1,SQRT(3))->PP3; CONSPOINT(3,3)->PP4;
      CONSTRIANGLE(PP1,PP3,PP2)->TR1;
      CONSTRIANGLE(PP3,PP4,PP2)->TR2;


      COMMENT
15.2; VARS ARR TO DEP FROM CODE DESTFLIGHT CONSFLIGHT FLIGHTS C;

      RECORDFNS("FLIGHTS",[0 0 0 0 0 ])->ARR ->TO ->DEP ->FROM ->CODE
                                          ->DESTFLIGHT ->CONSFLIGHT;
      CONSFLIGHT->C;
      [% C(1,"EDIN",0100,"LOND",0200),
         C(2,"LOND",0210,"PRIS",0310),
         C(3,"LOND",0210,"BERL",0310),
         C(4,"EDIN",0030,"BERL",0150),
         C(5,"BERL",0300,"MOSC",0500)  %]->FLIGHTS;
```

```
FUNCTION GETFROM P1 T1 P2 T2 FLIGHTLIST;
  VARS FLIST GF FHD; FLIGHTLIST->FLIST;
  IF P1=P2 AND T1<T2 THEN NIL EXIT;
LOOP: IF FLIST.NULL THEN "FAIL" EXIT;
      FLIST.NEXT->FLIST ->FHD;
      IF FHD.FROM=P1 AND FHD.DEP>T1
        THEN GETFROM(FHD.TO,FHD.ARR,P2,T2,FLIGHTLIST)->GF;
             IF NOT(GF="FAIL") THEN (FHD.CODE)::GF; RETURN
             CLOSE
      CLOSE
      GOTO LOOP
END;


     COMMENT
16.1; FUNCTION MEMBER XX XXL;
          IF XXL.NULL THEN FALSE
        ELSEIF XX=XXL.HD THEN TRUE
                         ELSE MEMBER(XX,XXL.TL)
          CLOSE
      END;


VARS VARLIST; [X Y Z]->VARLIST;

FUNCTION PDIFF E VAR;
      IF E.ISNUMBER THEN 0
  ELSEIF E.ISWORD THEN     IF E=VAR THEN 1
                       ELSEIF MEMBER(E,VARLIST)
                            THEN 0
                            ELSE DIFFERROR(E)
                          CLOSE
  ELSEIF E.DATAWORD="SUM"
         THEN PDIFF(SUM1(E),VAR)++PDIFF(SUM2(E),VAR)
  ELSEIF E.DATAWORD="PROD"
         THEN PROD2(E)**PDIFF(PROD1(E),VAR)
              ++ PROD1(E)**PDIFF(PROD2(E),VAR)
  ELSEIF E.DATAWORD="EXP"
         THEN EXP2(E)**EXP1(E)↑↑(EXP2(E)-1)**PDIFF(EXP1(E),VAR)
      CLOSE
 END;


FUNCTION MAKEEXP2 E1 E2;
      IF NOT(E2.ISNUMBER) THEN DIFFERROR(E2)
  ELSEIF E2=0 THEN 1
  ELSEIF E2=1 THEN E1
               ELSE CONSEXP(E1,E2)
      CLOSE
END;


MAKEEXP2->NONOP ↑↑;


     COMMENT
16.2; FUNCTION EVAL E N;
          IF E.ISNUMBER THEN E
      ELSEIF E ="X" THEN N
      ELSEIF E.DATAWORD="SUM"
           THEN EVAL(SUM1(E),N)+EVAL(SUM2(E),N)
      ELSEIF E.DATAWORD="PROD"
           THEN EVAL(PROD1(E),N)*EVAL(PROD2(E),N)
      ELSEIF E.DATAWORD="EXP"
           THEN EVAL(EXP1(E),N)↑EXP2(E)
          CLOSE
      END;
```

```
FUNCTION TABKDIFF E K A B DELTA;
LOOP1: IF K>0 THEN DIFF(E)->E;
                   K-1->K;
                   GOTO LOOP1
       CLOSE;
LOOP2: IF A=<B THEN NL(1); PR(A);
                    SP(5); PR(EVAL(E,A));
                    A+DELTA->A;
                    GOTO LOOP2
       CLOSE
END;


COMMENT
16.3; VARS SENTLIST; NIL->SENTLIST;

FUNCTION NOTT SENT;
  "FA678LSE"::SENT
END;

FUNCTION ADDSENT SENT;
  IF SENT.HD="FA678LSE"
    THEN (SENT.TL::FALSE)::SENTLIST->SENTLIST;
    ELSE (SENT::TRUE)::SENTLIST->SENTLIST
  CLOSE
END;

FUNCTION LOOKUP SENT LIST EQFN;
      IF LIST.NULL THEN UNDEF
  ELSEIF EQFN(SENT,LIST.HD.FRONT) THEN LIST.HD.BACK
                                  ELSE LOOKUP(SENT,LIST.TL,EQFN)
      CLOSE
END;

FUNCTION LISTEQ L1 L2;
      IF L1.NULL AND L2.NULL THEN TRUE
  ELSEIF L1.NULL OR L2.NULL THEN FALSE
  ELSEIF L1.HD=L2.HD THEN LISTEQ(L1.TL,L2.TL)
                     ELSE FALSE
      CLOSE
END;

FUNCTION TVALOF SENT;
  LOOKUP(SENT,SENTLIST,LISTEQ)
END;

VARS P2OR P1OR DESTOR OPERATION 2 ORF;
RECORDFNS("OR",[0 0])->P2OR ->P1OR ->DESTOR ->NONOP ORF;
VARS P2AND P1AND DESTAND OPERATION 2 ANDF;
RECORDFNS("AND",[0 0])->P2AND ->P1AND ->DESTAND ->NONOP ANDF;
VARS P2IMP P1IMP DESTIMP OPERATION 1 IMPF;
RECORDFNS("IMP",[0 0])->P2IMP ->P1IMP ->DESTIMP ->NONOP IMPF;
VARS P1NOT DESTNOT OPERATION 3 NOTF;
RECORDFNS("NOT",[0])->P1NOT ->DESTNOT ->NONOP NOTF;

FUNCTION ORFUN P1 P2;
      IF P1=TRUE  OR  P2=TRUE  THEN TRUE
  ELSEIF P1=FALSE AND P2=FALSE THEN FALSE
                               ELSE UNDEF
      CLOSE
END;

FUNCTION ANDFUN P1 P2;
      IF P1=TRUE AND P2=TRUE  THEN TRUE
  ELSEIF P1=FALSE OR P2=FALSE THEN FALSE
                              ELSE UNDEF
      CLOSE
END;
```

```
FUNCTION NOTFUN P1;
  IF P1=UNDEF THEN UNDEF
              ELSE NOT(P1)
  CLOSE
END;

FUNCTION IMPFUN P1 P2;
  ORF(NOTF(P1),P2)
END;

FUNCTION TVAL P;
        IF P.DATAWORD="PAIR" THEN TVALOF(P)
  ELSEIF P.DATAWORD="NOT"
        THEN NOTFUN(TVAL(P1NOT(P)))
  ELSEIF P.DATAWORD="OR"
        THEN ORFUN(TVAL(P1OR(P)),TVAL(P2OR(P)))
  ELSEIF P.DATAWORD="AND"
        THEN ANDFUN(TVAL(P1AND(P)),TVAL(P2AND(P)))
  ELSEIF P.DATAWORD="IMP"
        THEN IMPFUN(TVAL(P1IMP(P)),TVAL(P2IMP(P)))
      CLOSE
END;


      COMMENT
17.1  THE FUNCTIONS ASKED FOR ARE ISWIN AND PLAY;
      FUNCTION LINE I DI J DJ BOARD FUN;
        VARS S1 S2 S3;
        BOARD(I,J)->S1;
        BOARD(I+DI,J+DJ)->S2;
        BOARD(I+2*DI,J+2*DJ)->S3;
        FUN(S1,S2,S3);
      END;

FUNCTION APPLINES BOARD FUN;
  VARS ROW COL X;
  1->ROW;
ROWS: IF LINE(ROW,0,1,1,BOARD,FUN) THEN ->X;
                                 [%ROW,X%] RETURN

  ELSEIF ROW<3 THEN ROW+1->ROW; GOTO ROWS
      CLOSE;
  1->COL;
COLS: IF LINE(1,1,COL,0,BOARD,FUN) THEN ->X;
                                 [%X,COL%] RETURN
  ELSEIF COL<3 THEN COL+1->COL; GOTO COLS
      CLOSE;
DIAGS: IF LINE(1,1,1,1,BOARD,FUN) THEN ->X; [%X,X%] RETURN
   ELSEIF LINE(3,-1,1,1,BOARD,FUN) THEN ->X; [% 4-X,X %] RETURN
                        ELSE FALSE
      CLOSE
END;

FUNCTION WIN S1 S2 S3;
  IF S1=S2 AND S2=S3 AND NOT(S1="BLANK") THEN S1,TRUE
                                         ELSE FALSE
  CLOSE
END;

FUNCTION GOOD S1 S2 S3 CHAR;
       IF S1=CHAR AND S2=CHAR AND S3="BLANK" THEN 3,TRUE
  ELSEIF S1=CHAR AND S2="BLANK" AND S3=CHAR THEN 2,TRUE
  ELSEIF S1="BLANK" AND S2=CHAR AND S3=CHAR THEN 1,TRUE
                                            ELSE FALSE
      CLOSE
END;

FUNCTION GOODO S1 S2 S3;
  GOOD(S1,S2,S3,"NOUGHT");
END;
```

```
FUNCTION GOODX S1 S2 S3;
  GOOD(S1,S2,S3,"CROSS");
END;


FUNCTION OGOOD BOARD;
  APPLINES(BOARD,GOODO)
END;


FUNCTION XGOOD BOARD;
  APPLINES(BOARD,GOODX)
END;


FUNCTION BLINE S1 S2 S3;
  IF S1="BLANK" THEN 1,TRUE
  ELSEIF S2="BLANK" THEN 2,TRUE
  ELSEIF S3="BLANK" THEN 3,TRUE
                        ELSE FALSE
      CLOSE
END;


FUNCTION BLANK BOARD;
  APPLINES(BOARD,BLINE)
END;


FUNCTION ISWIN BOARD;
  VARS ANSWER;
  APPLINES(BOARD,WIN)->ANSWER;
  IF NOT(ANSWER=FALSE) THEN IF ANSWER.HD.ISNUMBER
                              THEN ANSWER.TL.HD
                              ELSE ANSWER.HD
                            CLOSE;
                        ELSE FALSE
  CLOSE
END;


FUNCTION PRN SQUARE;
      IF SQUARE="CROSS" THEN PR("X")
  ELSEIF SQUARE="BLANK" THEN PR(".")
  ELSEIF SQUARE="NOUGHT" THEN PR("O")
      CLOSE
END;


FUNCTION DISPLAY BOARD;
  VARS ROW COL; 1->ROW; 1->COL;
  NL(1);
LOOP: PRN(BOARD(ROW,COL));
           IF COL<3 THEN COL+1->COL; GOTO LOOP
      ELSEIF ROW=3 THEN  NL(1)
                   ELSE 1->COL; ROW+1->ROW;
                        NL(1); GOTO LOOP
           CLOSE;
END;


FUNCTION NEWBOARD;
  NEWARRAY([1 3 1 3],LAMBDA I J; "BLANK" END);
END;
```

```
FUNCTION PLAY BOARD;
  VARS IS OG XG BL;
  ISWIN(BOARD)->IS;
  IF NOT(IS=FALSE) THEN (IS::[HAS WON])=> EXIT;
  OGOOD(BOARD)->OG;
  IF NOT(OG=FALSE) THEN "NOUGHT"->BOARD(OG.HD,OG.TL.HD);
                        DISPLAY(BOARD);
                        [ I HAVE WON]=> EXIT;
  XGOOD(BOARD)->XG;
  IF NOT(XG=FALSE) THEN "NOUGHT"->BOARD(XG.HD,XG.TL.HD);
                        DISPLAY(BOARD);
                        [ YOUR TURN]=> EXIT;
  BLANK(BOARD)->BL;
  IF NOT(BL=FALSE) THEN "NOUGHT"->BOARD(BL.HD,BL.TL.HD);
                        DISPLAY(BOARD);
                        [YOUR TURN]=> EXIT;
  [ITS A DRAW]=>
END;

      COMMENT
17.2; FUNCTION ELEMENT I K M N P;
        VARS SUM J; 0->SUM; 1->J;
      LOOP: M(I,J)*N(J,K) + SUM ->SUM;
            IF J=P THEN SUM
                   ELSE J+1->J; GOTO LOOP
            CLOSE
      END;

      FUNCTION MULTARR M N P;
        NEWARRAY([% 1,P,1,P %],LAMBDA I K;ELEMENT(I,K,M,N,P) END);
      END;


      COMMENT
17.3; FUNCTION ORDER INTARR SIZE;
        VARS CHANGES N;
      PASS: 0->CHANGES; 1->N;
      THRU: IF INTARR(N)>INTARR(N+1)
              THEN INTARR(N),INTARR(N+1)->INTARR(N) ->INTARR(N+1);
                   CHANGES+1->CHANGES;
            CLOSE;
            N+1->N;
                IF N<SIZE THEN GOTO THRU
            ELSEIF CHANGES>0 THEN GOTO PASS
                CLOSE
      END;


      COMMENT
18.1; FUNCTION CHLSTRING CHLS;
        VARS N STRING;
        INITC(LENGTH(CHLS))->STRING;
        1->N;
      LOOP: IF NULL(CHLS) THEN STRING EXIT
            CHLS.NEXT->CHLS->SUBSCRC(N,STRING);
            N+1->N;
            GOTO LOOP
      END;


      COMMENT
18.2; FUNCTION COPYS S1 S2 N;
        VARS J L1;
        DATALENGTH(S1)->L1;
        1->J;
      LOOP: SUBSCRC(J,S1) -> SUBSCRC(J+N,S2);
            IF J = L1 THEN EXIT
            J+1->J;
            GOTO LOOP
      END;
```

```
        FUNCTION JOIN S1 S2;
          VARS S12;
          INITC(DATALENGTH(S1)+DATALENGTH(S2))->S12;
          COPYS(S1,S12,0);
          COPYS(S2,S12,DATALENGTH(S1));
          S12
        END;

        VARS OPERATION 2 <->;
        JOIN->NONOP <->;


        COMMENT
18.3;   VARS S S2;
        INIT(100)->S;

        FUNCTION A I J;
          SUBSCR(I+10*(J-1),S)
        END

        LAMBDA X I J;
          X->SUBSCR(I+10*(J-1),S)
        END->UPDATER(A);

        INIT(55)->S2;

        VARS OPERATION 5 ///;
        LAMBDA AN BN; AN//BN->AN->BN; AN; END->NONOP ///;

        FUNCTION A2 I J;
          IF I<J THEN 0
                 ELSE SUBSCR(I+10*(J-1)-J*(J-1)///2,S2);
          CLOSE
        END;

        LAMBDA X I J;
          IF I<J THEN "ERROR"=> RETURN
                 ELSE X->SUBSCR(I+10*(J-1)-J*(J-1)///2,S2);
          CLOSE
        END ->UPDATER(A2);


        COMMENT
19.1;   VARS SQRTS;
        MAPLIST(%SQRT%)->SQRTS;


        COMMENT
19.2;   FUNCTION MEMBER XX XXL;
               IF XXL.NULL THEN FALSE
          ELSEIF XX=XXL.HD THEN TRUE
                           ELSE MEMBER(XX,XXL.TL)
             CLOSE
        END;

        VARS MEMLODD;
        MEMBER(%[ 2 3 5 7 11 13 17 19]%)->MEMLODD;


        COMMENT
19.3    ** 99
        ** 8
        ;


        COMMENT
19.4A;FUNCTION FUNPROD F G;
          LAMBDA X F1 G1; G1(F1(X)) END(%F,G%)
        END;
```

```
      VARS OPERATION 3 **;
      FUNPROD->NONOP **;

      COMMENT
19.4B [% SIN(COS(S)),SIN(COS(Y)),SIN(COS(Z)) %] ;

      COMMENT
19.4C THE ANSWER IS HD(TL(TL([1 2 3]))) I.E. 3;
      VARS OPERATION 4 &;
      APPLY->NONOP &;

      FUNCTION TWICE F;
        LAMBDA X F; F(F(X)) END(%F%)
      END;


      COMMENT
19.4D YIELDS TRUE IF THE HEAD OF ITS ARGUMENT IS "MONKEY" ;

      COMMENT
19.4E ANOTHER PREDICATE;


      COMMENT
19.5; FUNCTION MAKETIMEBOMB N;
        VARS BOMB;
        LAMBDA ZERO TIME FUSE SELF;
          IF TIME=ZERO AND FUSE="SET"
            THEN NL(6); PR("EXPLODE"); NL(6)
          CLOSE;
          TIME+1->FROZVAL(2,SELF)
        END(%N,1,"SET",UNDEF%)->BOMB;
        BOMB->FROZVAL(4,BOMB);
        BOMB
      END;


      FUNCTION DEFUSE ABOMB;
        "UNSET"->FROZVAL(3,ABOMB)
      END;



      COMMENT
19.6; FUNCTION MAPLIST2 LIST FN;
        VARS BILL;
        LAMBDA LIST1 FN1 SELF;
          IF NULL(LIST1) THEN TERMIN
                         ELSE FN1(LIST1.HD);
                              LIST1.TL->FROZVAL(1,SELF)
          CLOSE
        END(%LIST,FN,UNDEF%)->BILL;
        BILL->FROZVAL(3,BILL);
        FNTOLIST(BILL)
      END;



      COMMENT
20.1; VARS PRL; PRINTRL(%2,3%)->PRL;

      FUNCTION PLINE X EXPR;
        VARS Y;
        SP(1); PR(X); SP(2);
      LOOP: PRL(EXPR(X,Y));
            IF Y<8 THEN Y+1->Y; GOTO LOOP
                   ELSE NL(1);
            CLOSE
      END;
```

```
FUNCTION PR08 EXPR;
  VARS X; 1->X;
  NL(4); SP(2); PR("*"); PR("Y"); SP(2); PR(0);
LOOP1: SP(5); PR(X);
       IF X<8 THEN X+1->X; GOTO LOOP1
              ELSE NL(1); SP(2); PR("X"); NL(1);
       CLOSE;
  0->X;
LOOP2: PLINE(X,EXPR);
       IF X<8 THEN X+1->X; GOTO LOOP2
              ELSE NL(4);
       CLOSE;
END;

PR08(%LAMBDA X Y;SQRT( X*X + Y*Y) END%)->PR08;


      COMMENT
20.2; VARS EVENREP;
      LAMBDA N;
        N+2->FROZVAL(1,EVENREP);
        N
      END(%0%)->EVENREP;


      COMMENT
20.3; FUNCTION SORTREP INTREP;
        LAMBDA INTREP1;
          SQRT(INTREP1());
        END(%INTREP%)
      END;

      FUNCTION SUMREP INTREP;
        LAMBDA INTREP1;
          INTREP1() + INTREP1()
        END(%INTREP%)
      END;


      COMMENT
20.4; FUNCTION PRINTPROG R C;
        LAMBDA R1 C1;
          VARS HELLOJIM;
          R1()->HELLOJIM;
          C1(HELLOJIM);
          HELLOJIM
        END(%R,C%)
      END;


      COMMENT
      NO EXERCISES IN SECTION 21;


      COMMENT
22.1; VARS PFUN; TRUE->PFUN;

      MACRO P1FUNCTION;
        VARS FNNAME VARNAME;
        IF PFUN.NOT THEN MACRESULTS([FUNCTION]) EXIT;
        ITEMREAD()->FNNAME;
        ITEMREAD()->VARNAME;
        ERASE(ITEMREAD());
        MACRESULTS
        ([% "FUNCTION",FNNAME,VARNAME,";",VARNAME,"=>" %]);
      END;
```

```
        MACRO PFUNCTION;
          VARS FNNAME THIS FIRST SECOND;
          IF PFUN.NOT THEN MACRESULTS([FUNCTION]) EXIT;
          NIL->SECOND;
          [% "FUNCTION",ITEMREAD() %]->FIRST;
        LOOP: ITEMREAD()->THIS;
              IF THIS=";"
                THEN MACRESULTS(FIRST<>[;]<>SECOND) RETURN
              CLOSE;
              FIRST<>[% THIS %]->FIRST;
              SECOND<>[% THIS,"=>" %]->SECOND;
              GOTO LOOP
        END;


        COMMENT
22.2:   MACRO ->>;
          VARS X Y Z;
          ITEMREAD()->X;
          ERASE(ITEMREAD());
          ITEMREAD()->Y;
          ERASE(ITEMREAD());
          ITEMREAD()->Z;
          MACRESULTS([% "->",X,";",X,"->",Y,";",Y,"->",Z %])
        END;


        COMMENT
22.3:   FUNCTION EVALUATE;
          VARS EX X;
          LISTREAD()->EX;
          EX<>[; GOON]->EX;
          NL(1); SP(1); PR("X"); SP(5); PR("EXPR"); NL(1);
          1->X;
        LOOP: PR(X); SP(5);PR(POPVAL(EX)); NL(1);
              IF X<10 THEN X+1->X; GOTO LOOP CLOSE
        END;


        COMMENT
        NO EXERCISES IN SECTION 23;


        COMMENT
        NO EXERCISES IN SECTION 24;
```