PROGRAMMING IN POP-2

R. M. Burstall
J. S. Collins
R. J. Popplestone

at the University Press
Edinburgh

22 George Square, Edinburgh
Printed in Great Britain by
Unwin Brothers Limited
The Gresham Press, Old Woking, Surrey.

ISBN 0 85224 197 6

CONTENTS

PREFACE

POP-2 is a programming language designed by R. M. Burstall and R. J. Popplestone and based on R. J. Popplestone's POP-1 (1968). POP-2 differs from most programming languages because it is designed for non-numerical as well as numerical applications. In addition, POP-2 is a conversational language allowing the user to communicate with his program and vice versa while it is running. This conversational property enables POP-2 to be used as a rather powerful calculating machine as well as a conventional computing system. A combination of these two modes produces a tool which can be matched to the particular problem being solved.

The first part of this publication is a quick guide to the aims and features of POP-2.

The Primer of POP-2 Programming, forming the second part, acquaints the reader with the POP-2 language and its terminology. Not all details of the language are described; enough is described, however, to provide the reader with a solid foundation in the language so that further questions about it can be directed to the Reference Manual. Although no previous experience of programming languages is required on the part of the reader, any such experience will be an advantage.

The Reference Manual, forming the third part, is a precise definition of the POP-2 language. Although the Primer can be read in its entirety before looking at the Reference Manual, it is recommended that the two parts are read in conjunction with each other. Thus, having read a section of the Primer on, say, conditionals, the corresponding section of the Reference Manual should be studied.

The final part consists of descriptions and listings of programs from the software library, mostly written at Edinburgh, and tested on the ICL 4100 system there. It illustrates the scope of the language and displays a number of programming tools and techniques. Examples are DEBUG—a debugging and tracing aid—and EASYFILE, which provides a disc filing system for programs and data.

The Reference Manual was originally published in *Machine Intelligence 2* (Edinburgh University Press 1968) and then republished with a brief introduction in *POP-2 Papers* by Oliver and Boyd (later distributed by Edinburgh University Press).

Since the reference manual was completed two years ago a number of errors, ambiguities, and omissions have come to light, and further experience of using the language has shown the need for a few minor changes and two additions of some significance. The first of these additions is *jumpout*, a facility which allows immediate exit from execution of one or more nested function bodies (see also the note on p. 279 on an extension giving generalized jumps and back-tracking). The second is the ability to break the program down into *sections*, somewhat analogous to ALGOL blocks, thus preventing clashes of identifiers.

The significant changes are listed in Appendix 3 of the Reference Manual. Many existing POP-2 programs should run unchanged and the remainder will need only a few simple alterations.

The 'Introduction to POP-2' which originally accompanied the reference manual, was brief and covered only the main points of the language. It has now been rewritten and greatly expanded to appear here as 'A primer of POP-2 programming'.

Acknowledgements

The suggestions and criticism of many members of the Department of Machine Intelligence and Perception at Edinburgh University, especially Mr Ray Dunn, and of others outside it, especially Mr Michael Healy and Mr Michael Woodger and those who have implemented POP-2 systems, are gratefully acknowledged. Technical consultations with Dr Michael Foster and Dr David Park about storage control were valuable. Thanks are due to Mrs Pat Ambler who has helped in the editing of this book, to Mr Bruce Anderson who has provided answers to the exercises in the primer and made many useful comments, and to Miss Eleanor Kerse who typed the new manuscript through numerous drafts.

POP-2 has been implemented on the ICL 4100 by Messrs Robin Popplestone, Ray Dunn, and David Pullin, on the ICL 1900 by Mr John Scott, and on the ICL System-4 and IBM 360 by Mr John Barnes and Mr Rod Steel (using a machine independent version of the compiler written in POP-2). The library of programs has been built up by many hands and organized by Mr Ray Dunn and Mr Robert Owen. The work has been part of a machine intelligence project, directed by Professor Donald Michie, whose guidance and encouragement have been invaluable, and supported by Edinburgh University, the Science Research Council, and the Medical Research Council.

We would like to thank Edinburgh University Press for their painstaking work on a very technical book, especially Dr Helen Muirhead, whose care and patience never failed us.

R. M. Burstall (Editor)

PART 1. A QUICK GUIDE TO THE MAIN FEATURES OF POP-2

R. M. BURSTALL AND R. J. POPPLESTONE

SUMMARY

POP-2 is a new computer language. Conceptual affinities can be traced to
1. John McCarthy's LISP (1962), from which it takes ideas for handling non-numerical objects of computation (lists).
2. Christopher Strachey's CPL (1963) and Peter Landin's ISWIM (1966), which foreshadow the aim of making a programming language into a notation with full mathematical generality, akin to algebra.
3. Cliff Shaw's JOSS (1964), which it resembles in its 'conversational' facilities.
4. Robin Popplestone's POP-1 (1968) of which POP-2 represents a rationalized and greatly-extended development.

These ingredients have produced a powerful but compact language for non-numerical programming. POP-2 was designed for implementation on a medium-sized machine with a modest investment in system programming. Because the language had to be stripped down to the level of the basic mathematical principles of programming, it is unrestrictive and open-ended.

The main distinctive features of POP-2 are
1. The syntax is very simple but the programmer has some freedom to extend it.
2. The programmer can create a wide variety of data structures: words, arrays, strings, lists, and records. A 'garbage collector' automatically controls storage for him.
3. Functions can be used in the same manner as in mathematics or logic, for example, as arguments or results of other functions, with no unfortunate restrictions on free variables.
4. The novel device of 'partial application' allows one to fix the value of one or more parameters of the function. This has a surprising multiplicity of uses, for example, to generalize the notion of an array to non-numerical subscripts and to disguise the distinction between stored values and computed values.
5. Another technique, 'dynamic lists', enables a physical device like a paper tape reader to be treated as if it were an ordinary list.
6. The programmer can call for immediate execution of statements at any time, giving facilities for conversational use and rapid debugging of complex programs.
7. The facility for immediate execution together with the variety of data structures available makes POP-2 suitable for use as the control language of a time-sharing system, enabling the user to effect filing, editing, compilation, and execution.
8. In the context of the widespread shortage of system programmers, a crucial feature is the open-endedness of the language. Work normally done in machine code by highly-skilled system programmers can be done in POP-2 itself.
9. POP-2 is compact and easy to implement. On the ICL 4100, for example, the whole system for compiling, running, and time sharing occupies only 22K of core (24-bit words). The effort needed to construct the complete system was less than 5 man-years. A machine-independent POP-2 in POP-2 compiler has been written.

A V A I L A B I L I T Y

POP-2 compilers are now available for the ICL 4100, ICL 1900, ICL System 4, and IBM Systems 360 series of machines. A PDP-10 compiler is being written. Multi-POP/4130 is a single-language system for a 64-K machine with disc, serving 8 simultaneous users. The other three implementations in their present form provide POP-2 programming in single-user mode, time-shared with batch operations

These systems are available to academic or research bodies from the Department of Machine Intelligence and Perception, and through arrangements with the National Research and Development Corporatic from Conversational Software Ltd. CSL will also contract, on suitable terms, to develop extended versions of the present systems, and also new POP systems for other machine ranges. Enquiries may be addressed to

POP-2 Enquiries
Department of Machine Intelligence and Perception
Forrest Hill
Edinburgh EH1 2QL.

or to

Conversational Software Ltd
Hope Park Square
Edinburgh EH8 9NW

PART 2. A PRIMER OF POP-2 PROGRAMMING

R. M. BURSTALL AND J. S. COLLINS

1. INTRODUCTION

Two important features of the POP-2 programming language distinguishing it from many other languages are its inherent ability to be used in an on-line mode and the fact that it is not restricted to numerical manipulations.

One way of using a computer to solve a problem is to specify the problem, write a program to solve the problem, keypunch the program and have it executed by the computer. This method of using a computer assumes that: the problem is well defined; an accurate program is available; the user is a perfect typist. In some cases, such as routine data processing, these conditions are met fairly easily. In many cases, however, such as for computing research or any other research problem, or when the user is not an experienced programmer, these conditions cannot easily be met. It is then necessary for a *dialogue* to take place between the user and the machine. In this dialogue, the computer is asked to perform some computation. Having studied the results, the user requests another computation. The dialogue continues in a series of steps, each of which depends on what has happened up to that point.

In this situation, the user must be able to request the computer to carry out tasks without having to specify them all at the start of the session. POP-2 is a language designed for this kind of use. A fundamental property of the language is the ease with which the language can be extended in *ad hoc* directions.

Using a calculating machine is clearly an on-line activity with alternate action by the user and the machine. A notable deficiency of the calculating machine is that it can only execute one step at a time, so that the user is forced to interact with the machine even when he knows what the next step will be. Extending the POP-2 language is like adding extra keys to a calculating machine and attaching to them a meaning defined in terms of existing operations.

NON-NUMERICAL COMPUTING

Most programming languages fall into the class of either commercial or scientific programming languages. COBOL, a well-known example of a commercial programming language, facilitates the writing of programs to manipulate large quantities of information, such as payroll files. ALGOL and FORTRAN are well-known examples of scientific programming languages. Both are particularly suitable for engineering-type calculations where the bulk of the computing is simply arithmetic. The more recently introduced PL/1 attempts to meet both commercial and scientific requirements. None of these languages, however, is really suitable for writing programs to play chess, prove theorems, or carry out other complex non-numerical activities. This deficiency has long been felt and list-processing languages such as LISP, or text-processing languages such as COMIT have been developed for this type of application. Both LISP and COMIT are classed as non-numerical languages.

It is not easy to class POP-2 with the above languages. It has the basic numerical capability of ALGOL, the list-processing capability

of LISP, and elementary record-handling facilities similar to those of COBOL. POP-2 is not, however, just a mixture of these languages; it is essentially a simple language which includes the fundamental concepts of FORTRAN, ALGOL, LISP, and COBOL, and has the ability to add new features in a natural way. For example, it is easy to write a matrix-processing package in POP-2 which enables the user to write arithmetic expressions of any complexity involving matrices.

To get a quick idea of the flavour of the language it may be helpful to look at the example of POP-2 program text in section 1.3 of the Reference Manual (see Part 3) and at some of the programs in the Program Library (see Part 4).

USING A POP-2 SYSTEM

The Reference Manual defines fully the POP-2 language. It does not, however, deal with problems like correcting typing errors, punching POP-2 programs, or getting permission to log into a POP-2 system; these are likely to vary from one installation to another. Each implementation of POP-2 for a particular computer system is described in a *functional specification* for the particular implementation. There are, therefore, as many functional specifications as there are different computer systems for which POP-2 has been implemented. Before any of the examples or exercises in this book can be tried out, the appropriate POP-2 functional specification must be consulted. It will describe what peripheral devices are available, how to log into the system, how the user is charged, and all such details.

A feature of most POP-2 systems is the *console*, through which communication takes place between the user and his program. This is usually a teleprinter, which allows the user to type his requests on the keyboard and the computer to print the results. Some POP-2 systems use another input-output mode, such as punched cards, as the main means of communication, but we will talk here as if a console were being used.

2. SIMPLE ARITHMETIC

The simplest use of a POP-2 system is as a rather high-powered calculating machine. Having logged into the system, the system is ready to execute any POP-2 statement we type. If we type the statement
2 + 2 => the answer ** 4 appears almost immediately. The *print arrow* sign => indicates that we wish the value of the preceding arithmetic expression to be printed. All results printed by this sign are preceded by the double asterisk.

The rules for writing numbers are very free. They are fully defined in sections 2.2 and 2.3 of the Reference Manual. Briefly, numbers may be integers or reals. Integers are written without a decimal point as a sequence of digits. Reals are written with a decimal point with at least one digit after the decimal point.

For example,
2.13 .2845 4.0
are legal reals but
4.
is not allowed.

Reals may have an *exponent part* consisting of the symbol $_{10}$ followed by a positive or negative integer. The integer is a power of ten by which the number is scaled.

For example,
$2.13_{10}1$ $.213_{10}2$ $213.0_{10}-1$ 21.3
all represent the same number.

ARITHMETIC EXPRESSIONS

The usual arithmetic operations + — * (for multiply) and / (for divide) are available. Arithmetic expressions involving these operations are evaluated following the usual rules of arithmetic; multiplication and division are carried out before addition and subtraction.

Thus if we type the statement
12.0 + 2.5 * 3.16 — 4
on the POP-2 console, the result
**15.9
is printed because the multiplication operation is carried out first. Notice that reals and integers can be mixed in arithmetic expressions.

Two further arithmetic operations are provided. The exponential operation ↑ enables a value to be raised to a power. For example, the arithmetic expression

(—2.5) ↑ 2

means minus two point five squared and produces the value 6.25. Similarly 4.0 ↑ 0.5 has the value 2.0.

An alternative division operation which may be used only between two integers is provided. This is written as //. This integer division operation produces two results; the quotient and the remainder.

Thus if the statement
25 // 3=>
is typed on the console, the results
**1 , 8
are printed, because 3 goes into 25 eight times with remainder one. Notice that the print arrow is able to print a sequence of results as well as just a single result.

PRECEDENCE AND PARENTHESES

Each of the operations described in the previous section has a *precedence* associated with it. A precedence is a number which determines the order in which the operations are applied. Both + and — have a precedence of 5. *, /, and // however have a precedence of 4, indicating that these operations are applied before addition and subtraction. The precedence table for arithmetic operations is as shown below.

Operation	Precedence
↑	3
* / //	4
+ —	5

Using this table, it can be seen that the result of typing the statement
3 — 2.5 ↑ 2 * 1.5 / 3 =>.
on the POP-2 console will be
**—0.125

The order of evaluation is as follows:

Original expression	3 — 2.5 ↑ 2 * 1.5 / 3
Apply operations of precedence 3	3 — 6.25 * 1.5 / 3
" " " " 4	3 — 3.125
" " " " 5	—0.125

In the case of operators of equal precedence they are applied from left to right, thus 6/2 * 5 = 15, not 0.6.

The precedence associated with each operation defines an order of evaluation of an expression. If another order is required, parentheses can be used in the conventional way.

For example, the statement
(3 — 2.5)↑2 * 1.5 / 3 =>
produces the result
**0.125

The rules of precedence apply to each expression within a pair of parentheses.

Thus the statement
(3 — 2.5↑2) * 1.5 / 3 =>
produces the result
**—1.625

(The number of figures printed after the decimal point may vary from one computer implementation to another.) Parentheses may be nested to any depth, the expressions within inner parentheses being evaluated first.

STANDARD FUNCTIONS

A number of mathematical functions are available for the POP-2 user. The list may vary from one implementation to another, so the functional specification for the particular implementation should be consulted for the precise list. The usual list is:

sin	trigonometric	sine,	angle in radians
cos	"	cosine,	" " "
tan	"	tangent,	" " "
arctan	"	arc tangent	" " "
sqrt	square root		
log	natural logarithm		
exp	"	anti-logarithm	

In POP-2, the argument of a function is enclosed in parentheses after the name of the function. So √2 is written *sqrt* (2).

Thus the POP-2 statement
exp(2 * *log*(1.414)) =>
produces the result
**2.0

The standard functions are listed at the end of the Reference Manual. So are some optional functions which an implementation of POP-2 on a particular computer may or may not provide (Appendix 2 of the Manual). Most implementations will provide a library of extra programs and functions with a simple method of compiling these, for example, *compile* (*library* (*[statistics]*)) might make available functions *mean*, *correlation*, *chisquare*, and so on, to do statistical tests. Part 4 'Program Library' gives descriptions and listings of many POP-2 library programs. They will be found to be of interest as examples

of POP-2 programming. The contents of the library may vary from one implementation to another.

EXERCISES

Answers are given as an appendix to this primer.

1. The arithmetic features of POP-2 have been described in this section. These facilities enable a POP-2 console to be used in a calculating machine mode. What would you type on a POP-2 console to evaluate the following arithmetic expressions?

(a) $\dfrac{2.5 \times 2}{-1.5 \times 4}$ (b) $1 + 2(5 - 3)$ (c) $\sqrt{3^2 + 4^2}$

(d) $sin^2 0.13 + cos^2 0.13$ (e) $tan^{-1} 1.5$

2. What is the value of the following POP-2 expressions?
(a) *8/2 * 6*
(b) $1 + 2.0_{10}-2 * 8$
(c) *7 * (sqrt (16) + 2)*

3. The following are not POP-2 expressions. Why not?
(a) *((2 * 3) + (4 — (6 + 3))*
(b) *sin 0.5*
(c) $1.25_{10}0.5$
(d) *6. + .5*

3. STATEMENTS, DECLARATIONS, AND VARIABLES

An *imperative* is a request to the POP-2 system to do something. An arithmetic expression followed by a print arrow is an imperative and requests that the expression be evaluated and the result printed. The imperative is actually carried out as soon as the print arrow is encountered. Thus, as well as being a printing operator, the print arrow is an imperative separator. The basic imperative separator is the semicolon and, in a sequence of imperatives, the individual imperatives must be separated from each other by a semicolon (or print arrow if appropriate).

There are two quite distinct types of imperative: the *declaration* and the *statement*. A declaration serves to introduce a new name or *identifier* by which some quantity will be known. The simplest kind of declaration introduces one or more *variables*. It consists of the word *vars* followed by the identifiers of the variables being introduced. For example, if we propose to use three variables called *x*, *y1* and *y2* then the declaration

vars *x y1 y2*

should be given. This declaration reserves space for the storage of three values. The three variables do not yet have any particular values. (Some POP-2 systems initialize them with values [*x. undef*], [*y1. undef*], and [*y2. undef*] respectively.) Identifiers can be made up of any group of letters and digits, beginning with a letter. Alternatively, they can be made up of any group of the following signs:

\+ — * / $ & = <> : £ ↑
For example: ++ —*— :: ↑:↑

If more than eight characters are used the extra ones are ignored: thus *variable1* is the same identifier as *variable2*.

Some identifiers are reserved as the names of standard variables, mostly those whose values are standard functions such as *sin*, *cos*, and *sqrt*. Others are reserved for syntactic purposes such as =>, :, —>, **end**. These are called *syntax words*. Syntax words such as **end** are printed in bold face in this book to remind the reader that they are so used, but in the actual POP-2 text they are not distinguished from any other identifier. Attempts to use reserved identifiers, for example, by writing **vars** *end*; are illegal.

A statement is an imperative that causes some computation to take place. An expression followed by a print arrow is an example of a statement. Another type of statement is an *assignment*, which enables a new value to be assigned to a variable. The assignment

```
2 +2 -> x
```

assigns the value 4 to the variable *x* replacing whatever was the previous value of *x*. Note that a semicolon must separate this statement from any statement or declaration preceding or following it. A typical imperative sequence might be:

```
vars x y1 y2;
2 +2 -> x;
x + 1 -> y1;
y1 =>
**5
y1↑2 -> y2;
y2 =>
**25
```

Notice that, assuming this imperative sequence is typed on the console keyboard, the imperatives are executed one by one as they are typed.

It is quite all right to write

```
x + 1 -> x;
```

This means that the new value of *x* is to be the old value plus one.

If a variable is used without declaring it first, it will be automatically declared and a message will be printed to indicate that this has happened.

Note that only variables may appear on the right-hand side of an assignment (a later section will indicate how certain kinds of expression may appear in this position too). It is not correct to execute assignments such as

```
x -> 2;
```

or

```
5 -> x + y;
```

The first is wrong because 2 is a constant—not a variable. The second assignment is incorrect because *x* + *y* is an expression.

Note that it is illegal to attempt to assign a new value to a standard variable, for example, 2 -> *sin*;

The reader may have wondered when, if ever, it is necessary to put in spaces or newlines. There is no distinction between a space and a newline, or between these and any sequence of spaces and newlines. They all serve to separate sequences of characters which might otherwise be confused. For example, if we want to write the identifier *x*2 followed by the number 3 we must write *x*2 3 not *x*23 which would form a single identifier, but 3*x*2 would be a number followed by an

identifier since it could not form a single identifier. Similarly /// —> *** is three identifiers whilst ///—>*** is one, but x —> y is the same as x—>y. In the case of real numbers spaces and newlines are not permitted in the middle of the number.

EXERCISES

1. Write assignments to exchange the value of two variables x and y. This can be done using a third variable.

2. What will be printed after typing in the following sequences of imperatives?

(a) **vars** *x1 x2*;
3 —> *x1*; *5* —> *x2*;
x1 + *x2* —> *x2*;
x1 + *x2* =>

(b) **vars** *a b c*
6 —> *a*; *7* —> *b*; *a* + *b* —> *c*;
c * *a* —> *b*;
a, *b*, *c* =>

3. Introduce a new temporary variable to write the following program more briefly and efficiently.

sqrt(*sin*(*x* + *a*)) * *exp*(*sin*(*x* + *a*)) =>

4. THE STACK

Consider the type of statement which consists of an arithmetic expression followed by a print arrow. This causes the arithmetic expression to be evaluated and the result to be printed on the console.

This process can be considered to take place in two stages:
(1) The arithmetic expression is evaluated.
(2) The result is printed.

In order that this can happen, the result obtained by evaluating the arithmetic expression must be left in some communication area for the print function to pick it up. This communication area is known as the *stack*. The stack is like a stack of cards on a table. When a new result is placed on the stack it becomes the new top of the stack and the first to be removed. The stack is therefore a *last-in-first-out* device.

Because the stack can accommodate more than one result, it is possible to write several expressions separated from each other by commas. For example,

2.5 + 3.6, 5 / 2, 2↑3
results in the three results 6.1, 2.5, and 8.0 being placed on the stack with 8.0 at the top.

The print arrow => does more than was implied in the previous section. It prints the entire stack, starting from the bottom, and empties it. Thus if the three expressions above were followed by a print arrow, the result

**6.1, 2.5, 8.0

would be printed and the stack would be left *empty*. Note that the top element of the stack is printed last. If we want to print just the top element instead of the whole stack we may write *pr*();.

We may write a whole sequence of statements which put numbers onto

the stack or remove them from the stack. Thus the sequence of four statements

1;2;—> y;—> x;

puts 1 on the stack, then 2 on top of it, then removes 2 assigning it to *y*, then removes 1 assigning it to *x*, leaving the stack in its original state. We have now met the following kinds of statement
(a) expression =>
(b) expression;
(c) expression —> variable;
(d) —> variable;
In any of these the expression may be replaced by a sequence of expressions separated by commas, and in (c) or (d) the part —> *variable* may be replaced by a sequence of variables each preceded by an arrow. Thus instead of the sequence above we could write

1,2 —> y —> x;

CAUTION. Leaving numbers around on the stack and using them later is an easy way to make mistakes. Do not do it wantonly.

The standard function *stacklength* which has no arguments tells the number of items on the stack. Thus we can discover this by typing

pr(stacklength());

The standard function *setpop*, also of no arguments, clears the stack.

POP-2 functions take their parameters, if any, from the top of the stack, and leave their results, if any, on the stack. Thus the function *sin* removes the top item from the stack, computes its sine, and places this number on the stack. When we write

sin(0.143)

0.143 is loaded on the stack and function *sin* is called. If we write
sin( )

whatever is currently on top of the stack is used by the function *sin*. If this expression were executed with the stack empty, an error message would be printed indicating that the stack had underflowed.

Provided the number of parameters (arguments) put on the stack when a function is called is the same number as taken by the function, any numbers previously on the stack are unaffected by the transaction.

Note the difference between writing
sin
which loads the sin function itself onto the stack and
sin( )
which actually causes the *sin* function to be executed. An alternative way of writing the latter is
.sin
which has an identical effect.

There is a standard function *erase* which takes one parameter and produces no result. *Erase* simply removes one item from the stack. Thus if we type

erase(2 + 2) =>

the expression in the parentheses will be evaluated but no result will be printed. A more useful example of *erase* is

erase (23//6) =>

which produces the result

**5

because the quotient, put on top of the stack by the integer division, is removed by the function *erase*.

EXERCISES

1. Assuming the functions *add* and *mult* replace the top two members of the stack with the sum and product respectively, what is left on the stack after executing the following?
2, 3, 4;add( ); mult( );

2. What is the effect of the following statements?
(a) *x, y —> x —> y;*
(b) *—> x —> y; x, y;*

3. Write a sequence of statements which exchange the first (top) and third items on the stack.

5. FUNCTION DECLARATIONS

Any POP-2 system will have a number of built-in standard functions such as square root. Facilities are provided to extend this basic set by defining new functions in terms of existing ones.

The *function definition:*
function *sumsq x y;*
x↑2 + y↑2
end

defines a new function called *sumsq* whose value is the sum of the squares of two numbers. *x* and *y* are called *formal parameters*. When the function is called, for example, by writing *sumsq(3, 4)*, these formal parameters will be assigned the values of the corresponding actual *parameters*, or arguments, that is, 3 and 4, before the expression in the body of the function definition is evaluated. Thus evaluating the expression *sumsq*(3, 4) causes the *body* of *sumsq*, that is, *x↑2 + y↑2* to be evaluated with *x* initialized (given an initial value) to 3, and *y* initialized to 4. The value 25.0 is left on the stack as a result.

Note that a function definition does not cause any calculation to be done; it simply creates a new function and assigns it as the value of a variable, in this case the variable *sumsq*. If the variable has not been previously declared, the function definition acts as a declaration of it. We must distinguish the act of defining a function from that of calling it, that is, applying it to some parameters, when the function is actually used to perform a calculation.

It is useful to know something about what takes place when a function is called. When the expression

sumsq(3, 4)

is evaluated, the values of the actual parameters 3 and 4 are placed on the stack. The piece of program associated with *sumsq* is then entered.

Because *sumsq* has two arguments, it takes two items off the stack and assigns them to *y* and *x*. (Note that *y* will be on top of the stack and will be the first to be removed.) The expression

x↑2 + y↑2

is then evaluated using these values of x and y and the result is left on the stack. The variables x and y belong to the function *sumsq* and have no connection with variables having the same name that might exist outside the function definition.

As a matter of fact we could achieve the same result as above without using this parameter mechanism defining *sumsq* in the following way:

```
function sumsq;
vars x y;
—> y; —> x                (take two numbers off the stack and assign them
x↑2 + y↑2                  to y and to x)
end
```

This produces the same effect because the parameter mechanism is defined to work in this way. The parameter mechanism is simply a shorthand notation for the above.

Notice that the *body* of a function definition, that is, the text occurring after the names of the function and its parameters, may be simply an expression, or it may be a sequence of statements. In either case it may include some declarations. It may also include the print arrow =>, so that calling the function may cause some values to be printed. If used inside a function, the print arrow => prints and removes *only the top item* on the stack.

If we have defined a function f and want to change it we simply redefine it. For example:

```
function f x;x + 1 end;
function f x;x + 2 end;
f(3) =>
**5
```

LOCAL VARIABLES

Consider the following piece of program:

```
vars a;2—> a;
function f x;vars a;
    x↑3 —> a;
    a + a
end;
f(3),a =>
```

The value printed for $f(3)$ is clearly 54.0, that is, $3^3 + 3^3$, but what has happened to a? Is it now 27.0, the value it assumed in the calculation of $f(3)$, or is it still 2, as it was originally? In fact the value printed will be 2, showing that the evaluation of $f(3)$ has not affected the value of a.

This is because one *identifier* can name more than one *variable*, and each variable may have a different value. When we write an identifier the context determines which variable is named by that identifier. When we pass through the declaration of an identifier a new variable is associated with that identifier. Thus **vars** a in the first line of the example above creates a new variable called a, and the **vars** a in the second line also creates a new variable called a whenever the function f is applied to some argument, for example, in the evaluation of $f(3)$. When $f(3)$ has been evaluated this new variable is no longer required, thus a variable declared in the body of a function ceases to exist when that body has been evaluated. Such a variable is called a *local* variable. When an identifier occurs in a statement it always denotes the variable which has most recently been associated with that identifier, excluding

any variables which have ceased to exist. For this purpose mentioning an identifier as a formal parameter, for instance, *x* in the example above, also acts as a declaration, and the variable it creates ceases to exist when the function body has been evaluated.

This convention enables us to use new identifiers freely to name formal parameters and local variables in a function body, knowing that what is done inside the function body cannot affect any variables outside which happen to have the same name.

Variables which are declared outside any function body (not formal parameters or local variables) are called *global* variables.

Just as the parameter list provides a convenient facility for declaring variables which are given values from the stack, so it is possible to declare local variables whose values are automatically placed on the stack when execution of the function is finished. Such a variable is called an *output local.* The function *sumsq* could be defined using an output local in the following way:

```
function sumsq x y => z;
x↑2 + y↑2 -> z
end
```

In this case, *z* is a local variable whose value is placed on the stack at the end of executing the function *sumsq.* Although the sign => is used to separate the parameters from the output locals, used in this context it has nothing to do with printing values. It is particularly convenient for functions which produce more than one result, for example,

```
function bothroots x => posroot negroot;
    sqrt(x) -> posroot; -posroot -> negroot
end;
bothroots(2) =>
**1.414, -1.414
```

If the results were merely left on the stack, instead of using output locals, the function would still work, but anyone reading its definition would have to look quite carefully to notice that it produces two results and to tell which comes first. Thus using output locals helps to make the program more readable, it is a matter of taste not necessity.

If we have already declared a variable as a local variable of a given function we may not declare it again as a local to the same function, unless the new declaration is inside some interior function.

Thus

```
function f x; vars a; vars a; ... end;
```

is illegal, but

```
function f x; vars a;
    function g y; vars a; ... end
    ...
end;
```

is all right.

If similar declarations occur twice globally, that is, outside any function body, the second one is simply ignored.

EXERCISES

1. Declare a function *roots* which takes three parameters *a*, *b*, and *c*, and produces, as results, the roots of the quadratic equation

$ax^2 + bx + c = 0$

using the expressions

$$\frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a}$$

If your first definition of *roots* does the same calculation twice, such as $\sqrt{(b^2 - 4ac)}$, rewrite it avoiding this duplication.

2. What is printed by the following program?

```
vars a b c; 1 --> a; 2 --> b; 3 --> c;
function f a => c; vars b;
    a * a --> b; b + b --> c
end;
f(a + b + c) + f(a + b + c) =>
```

3. Assume that the function *apply1ton* takes two arguments, the first an integer n and the other a function of one argument, and applies the function to all integers from 1 to n. For example,

```
function prsqrt n; sqrt(n) => end;
apply1ton(4, prsqrt);
**1.000
**1.414
**1.732
**2.000
```

How would you use this function to tabulate the value of the expression $(1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3)$ in the range 0 to 3.0 at intervals of 0.1? (The function itself is to be defined as an exercise in section 7.)

6. CONDITIONALS

It often occurs within a function definition that the particular result depends upon whether some condition is true or not. A *conditional* enables one of two possible courses of action to take place according to a condition.

Consider the function definition

```
function max x y;
if x > y then x else y close
end
```

The value of the expression $max(a, b)$ is a or b, whichever is greater. The operation > is an operation which produces a *truth value*. The two possible truth values are *false* and *true*. They are represented by the numbers 0 and 1 respectively, but for convenience the standard variables *false* and *true* always have these values.

Other operations which produce truth values are

```
>=  greater than or equal
=<  less than or equal
>   greater than
<   less than
=   equal to.
```

All relation operations have a precedence of 7. Thus comparison between arithmetic expressions will always take place after all arithmetic operations have been carried out. For example:

```
10 > 3 =>
**1
50 =< 20 =>
**0
true =>
**1
```

A conditional is always terminated with the syntax word **close**. Any imperative sequence (including conditionals) can appear between the **then** and the **else** and between the **else** and the **close**.

Quite complicated conditions can be tested using the syntax words **and** and **or** to join a series of conditions. For example,

if $x > 3$ **and** $y =< 2$ **then** $x + y$ **else** $x - y$ **close**

will leave $x + y$ on the stack if x is greater than 3 and y is less than or equal to 2. Otherwise the value of $x - y$ is left on the stack.

Note that **and** and **or** are not operations. They are syntax words associated with conditionals which simplify testing complicated conditions. They may only appear after **if** or **elseif** (see below) and before **then**. The conditionals joined by these syntax words are evaluated from left to right according to the following table (if there is no **else** then '**else** statement' means the statement after **close**).

basic word following cond	value of condition	what is evaluated next
and	false	**else** statement
and	true	next condition
or	false	next condition
or	true	**then** statement
then	false	**else** statement
then	true	**then** statement

The reason why **and** and **or** are not operations like * and +, and may not be used freely to form expressions, is that in the expression $x > 3$ or $y\uparrow 10 > 2$, for example, if x is greater than 3 there is no point in calculating $y\uparrow 10$ and comparing it with 2. Analogous functions are provided which do evaluate both arguments and may be used to form parenthesized expressions (see section 24 'Some useful standard functions'). The function *not* reverses a truth value so that we may write

if *not*(p) **or** *not*$(x = 3)$ **then ... close;**

A conditional can be used to select one of two possible imperative sequences or one of two possible expressions. In the first case, the conditional behaves like a statement and is usually called a *conditional statement.* In the second case it behaves like an expression, and is usually called a *conditional expression.*

Consider the following conditional expression:
if $x = 1$ **then** 2 **else**
if $x = 2$ **then** 4 **else**
if $x = 3$ **then** 3 **else** -1 **close close close**

Its value is 2, 4, 3, or −1 depending on the value of x. This structure occurs so often in programming that it is convenient to avoid having to write many **closes** at the end. This is achieved using the syntax word **elseif**, which behaves exactly like **else** followed by **if** except that no corresponding **close** is required. Using **elseif**, the above conditional expression may be rewritten:

if $x = 1$ **then** 2
elseif $x = 2$ **then** 4
elseif $x = 3$ **then** 3
else -1 **close**

which only has one **if** requiring a corresponding **close.**

Sometimes there is no action to be taken if the condition is false. In this case the else part may be omitted, for example,

if $x > 0$ **then** x => **close;**

EXERCISES

1. Rewrite the function *roots* defined in the exercise in section 5 to
(a) produce the complex roots if $b^2 - 4ac < 0$
(b) work correctly if $a = 0$.

2. Write a function to test whether a given number is less than 100 and divisible by 3, 4, or 5.

3. Tax is not levied on the first £150 of a man's income. It is levied at 10% on the next £250, at 25% on the next £200 and at 33% on the remainder. Write a function *tax* such that *tax* (*i*) is the tax levied on an income of £*i*.

4. Write a function which takes three parameters and puts them in ascending order of magnitude, for example,

f(*1*,*–2*,*4*) =>
** *–2*,*1*,*4*

7. LABELS AND GOTO STATEMENTS

In an imperative sequence, the statements are normally executed in the order in which they are written. The *goto statement* and its associated *label* enable statements to be executed in some other specified order. The destination of a goto statement is labelled with an identifier. The identifier (called a label) precedes the statement and is separated from it by a colon. After execution of a goto statement, the next statement to be executed is the one whose label is the destination of the goto statement.

Consider the following function definition:
function *tab fun x step hi;*
again: **if** *x* =< *hi* **then** *fun*(*x*) =>
x + *step* –> *x*; **goto** *again* **close**
end

This defines a function *tab* which tabulates the values of a function *fun* over a range from *x* to *hi* in steps of *step*. Executing the statement

tab(*sqrt*, *1*, *1*, *3*)
causes the following results to be printed:
***1.0*
***1.414*
***1.732*

Remember that if it is used in a function body => prints only the top item of the stack.

Goto statements and labels can only be used in a function definition, because labels are only appropriate if the labelled instruction is stored. Statements executed directly from the keyboard are not stored and cannot be labelled. Moreover a goto statement can only refer to a label in the same function definition as the goto statement itself. Clearly, to avoid ambiguity, no two statements may have the same label in a given function definition.

The goto statement which skips to the end of a function definition

occurs so frequently that a special form is provided which requires no label. The statement

return

terminates execution of the function in which it occurs and returns to the program calling the function exactly as if the last statement had been executed. The function *tab* could have been defined using **return** as follows:

```
function tab fun x step hi;
again: if x > hi then return close;
fun(x) => x + step -> x; goto again
end;
```

We could have put a label, say *finish*, before **end** and put **goto** *finish* instead of **return**. It is just that **return** is a little neater.

Another syntax word **exit** is provided. The word **exit** is identical to the pair of syntax words **return** followed by **close**. This pair occurs together quite frequently, and can always be replaced by the single word **exit**.

As well as illustrating the goto statement, the definition of the function *tab* above has some other features worth commenting on. The formal parameter *fun* is used to denote a function. The particular function is specified when the function *tab* is called. This ability to use a variable whose value is a function is an important property of POP-2. It means that it is easy to define functions which operate on functions and, as we shall see later, produce functions as results.

A further point to note about the definition of *tab* is the use of a formal parameter *x* as a variable whose value is changed during execution of *tab*. As *x* is a local variable which has merely been given the *value* of the actual parameter (via the stack), changing *x* cannot change any variables in the calling program. This means that if we defined a function *increment* as

```
function increment x; x + 1 -> x end
```

and called it by executing the statement

increment(*y*)

this would have no effect on *y* because the parameter given to *increment* is simply the current value of *y*. The effect is simply to declare a local variable *x*, give it the value of *y*, add 1 to it, and then lose the value on exit from the function. To increment *y* we may define

```
function increment; y + 1 -> y end
```

and call it by *increment*( ).

Many loops, that is, sequences of instructions which may be executed repeatedly, start off with a conditional. For example, in computing *n* factorial

```
l:  if i =< n then i*p -> p; i+1->i; goto l
    close;
```

To save making up a label and putting a statement to go back to it, we may replace **if** by **loopif**.

```
loopif i=<n then i*p->p; i+1->i
close;
```

In general we can always replace **if** by **loopif**, even when **elseif** and **else** are being used. As soon as a condition succeeds we jump back to **if** (we think of **else** as being preceded by the condition *true)*. Thus we have the equivalence

loopif ... **then** ...	*loop*:	**if** ... **then** ...; **goto** *loop*
elseif ... **then** ...		**elseif** ... **then** ...; **goto** *loop*
else ...		**else** ...; **goto** *loop*
close		**close**	

Here *loop* is any label which does not occur in any part of the same function definition. Of course this goes on for ever unless the dots contain some other **goto**, but so long as we leave out the **else** there is a way of stopping, since all the conditions may fail.

Counting on integers or real numbers is so common that a further abbreviation (in fact a standard 'macro', see section 22) is introduced to cope with the most common cases. Suppose *I* stands for any identifier and *M*, *K*, and *N* denote any identifiers or *unsigned* numbers. Then we may write

forall I M K N
to stand for
M—K—>I;
loopif (*I+K—>I; I=<N*) **then**

Thus *forall I M K N* means for all values of *I* from *M* up to *N* increasing in steps of *K*. The factorial loop above can be written simply as

```
forall i 1 1 n;
    i*p—>p
close;
```

If *K* is a real number we should take care over rounding errors.

forall x 0.0 0.1 1.01; ....
is safer than
forall x 0.0 0.1 1.00; ...

since 0.0 plus ten times 0.1 might come to, say, 1.0003 to within the computer's accuracy, and then the $x = 1.0$ value would not be done.

Remember that *forall* involves conditionals and **goto** so it can only be used inside a function body.

Straightforward loops can be done with *forall*. A more powerful and elaborate looping facility, the *FOR* facility, is provided in the Program Library (see Part 4).

We have not used **loopif** or **forall** in the answers to exercises of the Program Library since they were added during the revision of POP-2 and were not defined when this work was being done.

EXERCISES

1. The function *tab* used as an example in this section tabulates a function of one parameter over a specified range. Define a function *tab2* which tabulates a function of two parameters over specified ranges of the two parameters. Use *tab2* to print the products of all pairs of integers between 1 and 10. (For a way to get a proper tabular layout of the results see section 8 'Printing results'.)

2. The sequence
$x_0 = 1$
$x_{k+1} = \frac{1}{2}\left(x_k + \frac{n}{x_k}\right)$

gets closer and closer to the square root of N as k increases. Write a function *terms* such that *terms*(*n*, *epsilon*) is the value of k needed to compute the square root of n to within plus or minus epsilon. For example, if $n=9$ then $x_0=1, x_1=5, x_2=3.6, x_3=3.05$ and *terms*(9, 0.1) = 3.

3. Define the function *apply1ton* described in example 3 of section 5 'Function declarations'.

8. PRINTING RESULTS

So far we have used => to print results. This prints one or more results off the stack on a new line preceded by **.

Often we would like a different layout, and the following standard functions are provided (others are given in section 20).

pr(*x*) — this causes the value of x to be printed. Negative numbers are printed with a minus sign, positive ones preceded by a space.
prreal(*r*, *m*, *n*) — prints a real with m digits before the point and n digits after it.
nl(*k*) — this prints k new lines
sp(*k*) — this prints k spaces.

For example, to print the multiplication table:

```
function multab; vars i j p; 1 -> i;
    loopi: if i > 12 then exit;
           1 -> j; nl(1);
    loopj: if j > 12 then i + 1 -> i; goto loopi close;
           i * j -> p;
           sp(if p < 10 then 2 elseif p < 100 then 1 else 0 close);
           pr(p);
           j + 1 -> j; goto loopj
end;
```

multab():

```
 1   2   3   4   5   6   7   8   9  10  11  12
 2   4   6   8  10  12  14  16  18  20  22  24
 3   6   9  12  15  18  21  24  27  30  33  36
 etc.
```

Section 18 'Input and output facilities' gives further information about reading data and printing results, using the console or other input/output devices. Thus when we say above that something is printed we include the case where it is output to some other device, such as a disc file.

9. WORDS

So far, we have seen how a POP-2 variable can have a numerical value or a function value. Another important type of values is the *word*. A word, like an identifier, is made up of letters, digits, or signs. Only the first 8 characters are significant. Examples of words are

```
x    a5    pdp7    +    (
++    "    happy    birthday
```

The exact rules for constructing words are given in section 8.6 of the Reference Manual.

In order to assign a word to a variable, it is necessary to indicate that the word itself is meant, not the value of the variable designated by the word. For example, the assignment

cost —> *x*

assigns the *value* of the variable *cost* to the variable *x*. In order to assign the word *cost* it must be enclosed in quotes "and". In this case, the assignment would be

"*cost*" —> *x*

after which the value of *x* would be the word "*cost*" and the effect of printing *x* by typing

x =>

would be to print

***cost*

If the variable *c* has value 100 then
pr("*cost*");*pr*("=");*pr*(*c*);
prints
cost= *100*

We may test words for equality, for example, "*cat* " = "*cat*" has value *true*.

EXERCISES

1. What is printed by the following programs?

(a)
```
function out n w;
    nl(2);pr(n);sp(1);pr(w);pr(", ");pr("please")
end;
out(20, "pounds");
out(40,"dollars");
```

(b)
```
function truthval p;
    if p then "true" else "false" close
end;
truthval(50>40) =>
```

2. The standard function *destword* takes as its parameter any word and leaves on the stack integer representations of the characters and the number of characters in the word, for example, *destword("CAT")* leaves i_C, i_a, i_t, 3 on the stack where i_C, i_a, and i_t are the integers corresponding to *c*, *a*, and *t*. Define a function *order* which takes two words and produces "*before*", "*same*", *or* "*after*", depending on whether the first letter of the first word is before, the same as, or after the first letter of the second word. Assume that the integer representations of the letters are in consecutive ascending order.

10. LISTS AND LIST PROCESSING

A *list* is simply an ordered sequence of items. A list of words or positive integers or positive reals can most easily be constructed by enclosing the words, integers, or reals in square brackets. For example, the assignment

[*cat dog horse*] —> *x*

makes x a list of three words. The value of x is the whole list and if x is printed by executing

x =>

the result

** [*cat dog horse*]

is printed.

Another example of a list formed in this way would be

[*1 cat 2 dogs 3.1416 horses* ***]

Note that although *cat* is used as a word, not a variable, there is no need to enclose it in quotes since the brackets serve the same purpose as quotes.

There is a standard operation, written <>, which joins two lists together. Thus the expression

x <> [*donkey cow*]

produces a list like x but with 2 new items on the end. If this is now printed, the result would be

** [*cat dog horse donkey cow*]

The value of the variable x is not changed.

An item on a list may itself be a list. The list
[[*cat dog*][*horse donkey*]]
is a list of two items. The first item of the list is the list [*cat dog*]. The second item is the list [*horse donkey*].

There are two standard functions for accessing the items of a list. The function *hd* has as value the first item, or head, of the given list. Thus if x is the list [a b c d] the value of the expression

hd(x)

is the word "a". The function *tl* has as value the *tail* of the given list. The tail of a list is the list with the head removed. The value of the expression

tl(x)

is the list [b c d]. Thus the functions *hd* and *tl* can be used together to access any item on a list. The first item of the list x is *hd*(x), the second item is *hd*(*tl*(x)), the third item is *hd*(*tl*(*tl*(x))), and so on.

The tail of a list of one item is the word "*nil*", which represents the empty list. The standard variable *nil* has the word "*nil*" as its value. Any attempt to use the functions *hd* and *tl* on anything other than a list will result in an error. Thus an attempt to extract the third item from a list of two items by writing

[a b] –> x;
hd(*tl* (*tl* (x))) =>

results in an attempt to evaluate the expression

hd(*nil*)

which produces an error message because the word *nil* is not a list.

There is a standard function *null* whose value is *true* for an empty list and *false* otherwise. It is very frequently used to test for the end of a

list. Consider, for example, a possible definition of a function *length1*, the length of a list.

```
function length1 x; vars n;
0 -> n;
l1: if null(x) then n exit;
n + 1 -> n; tl(x) -> x; goto l1
end;
```

One way of constructing a list is to use square brackets. A more fundamental function is *cons* (short for construct) also written as ::, an operation of precedence 2. The expression *cons*(*a*,*b*), or *a* :: *b*, constructs a list whose head is *a* and whose tail is *b*. Thus

```
"cat" :: nil -> x;
```

makes *x* a list of one item—the word *cat*. A list of several items could be constructed using *cons* as follows:

```
"a" :: ("b" :: ("c" :: nil)) -> x
```

which creates exactly the same list as

```
[a b c] -> x
```

The latter is a shorthand notation for the first.

The function *cons* (i.e., ::) puts an item in front of a list. Let us define a function to put an item at the end of a list. This is

```
function append x xl; xl <> (x::nil) end;
```

Here *x* is an item and *xl* a list (we will make a habit of using identifiers such as *xl* and *yl* for lists), *x*::*nil* is the list whose only element is *x*, and the operation <> joins the list *xl* to the list *x*::*nil*. Thus

```
append(4, [1 2 3]) =>
** [ 1 2 3 4]
```

The reader may find the distinction between :: and <> and between *x* and *x*::*nil* a little puzzling. He may find the following picture helpful. Items are coloured beads and a list is a string with beads on it. The empty list *nil* is a string with no beads on it. If *x* is a bead and *xl* is a string of beads then *x*::*xl* puts an extra bead on the front of the string. If *xl* is a string of beads and *yl* is another string then *xl* <> *yl* ties the end of the first string to the beginning of the second. Thus if *x* is a bead *xl* <> *x* would be nonsense, since you cannot tie a bead onto the end of a piece of string, but *xl* <> (*x*::*nil*) is all right.

As a matter of fact this explanation is not strictly accurate. What happens if we do the following?

```
[2 3 4] -> x;
1::x -> y;
x =>
```

The answer should be [2 3 4] since it would be inconvenient if the second statement upset the result of doing the first. To ensure this *1*::*x* does not put the bead 1 onto the string called *x* but rather ties a fresh piece of string with the bead 1 on it in front of the string *x*, placing the beginning of this fresh piece in *y*. Similarly <> uses fresh string to copy its first argument so that the operation does not affect the value of other variables. The details will be described fully later on.

The following function tests whether an item occurs in a list.

```
function member x xl;
loop: if null(xl) then false
        elseif hd(xl) = x then true
        else tl(xl) -> xl; goto loop
      close
end;
member(1, [2 1 5]) =>
**1
member("Joe", [Fred Alf Bert]) =>
**0
```

The following pair of functions manipulate 'association lists', for example,
[*dog chien cat chat pig cochon*]

```
function assoc x xyl => y; vars xl;
loop: if null(xyl) then undef -> y
        else hd(xyl) -> xl; tl(xyl) -> xyl;
          if xl = x then hd(xyl) -> y
                    else tl(xyl) -> xyl; goto loop
          close
      close
end;
```

Note: *undef* is a standard variable with value "*undef*" meaning undefined.

```
function makeassoc x y xyl => xyl1;
    x::(y::xyl) -> xyl1
end;
```

Now we can use these in the following way

```
[dog chien cat chat pig cochon] -> dict;
assoc ("cat", dict) =>
** chat
makeassoc ("hen", "poule", dict) -> dict;
assoc ("hen", dict) =>
** poule
```

Suppose that we wish to obtain a new list, each member of which is derived from the corresponding member of some given list by applying a function to it. We could define a function *maplist* such that, for example, *maplist* ([*1 2 3 4*], *sqrt*) *is* [*1.00 1.41 1.73 2.00*]

```
function maplist xl f => yl; nil -> yl;
loop: if not(null(xl)) then append(f(hd(xl)), yl) -> yl;
                            tl(xl) -> xl; goto loop
                      close
end;
```

In fact *maplist* is a standard function.

EXERCISES

1. Given a function *p* which produces a truth value as its result, write a function *exists* such that *exists*(*xl*, *p*) is *true* just if *p* produces *true* for some element of *xl*.
2. Write a function *delete* such that *delete*(*x*, *xl*) is a list similar to *xl* but with any items equal to *x* on it deleted.
3. An association list *price* associates a price in pence with each of a number of articles. Write a function which will take a list of articles purchased and work out the total price (use *assoc*).

4. You are given a list, each of whose elements is an association list describing a known criminal thus

[[*name jones hair sandy eyes brown height 65*]
[*name crippen hair none eyes green height 61*]....]

Write a function which takes a specification of a wanted man, for example,

[*hair grey eyes brown height 60*],

and produces a list of the names of known criminals who might correspond to the description.

5. An association list is given which associates with each town a list of other towns which can be reached from it by a direct flight. Write a function to produce a list of all the towns which can be reached from a given one with not more than 1 change. Now write one for not more than *n* changes. (*Hint.* A function to remove repeated elements from a list would be useful, for example, *prune* ([1 2 3 2 5 3]) = [1 2 3 5].)

11. LAMBDA EXPRESSIONS

It was mentioned in the section on functions that variables can have functions as values, as well as the more obvious types of values such as numbers or truth values. A function definition is therefore a kind of assignment in which a function value is assigned to a variable. It is possible in POP-2 to write function constants just like we can write numerical constants (for example, 0, 3.15) or truth values (*true, false*). A function constant is called a *lambda expression* and is simply a way of defining a function and leaving the definition on the stack. The function *sumsq* defined earlier in the conventional way could have been defined as follows:

vars *sumsq*;
lambda *x y*; *x*↑*2* + *y*↑*2* **end** –> *sumsq*;

Thus the basic word **lambda** is very similar to the basic word **function** except that no function name is included. A lambda expression is an 'anonymous' function.

Lambda expressions are useful in a variety of circumstances. A frequently-occurring situation is illustrated by the following. We wish to use the *tab* function defined above to tabulate the values of *x*↑3 between 1 and 10 in steps of 0. 5.

We cannot write

tab(*x*↑*3*, *1*, *0. 5*, *10*);

because *tab* assumes the value of the first parameter is a function, whereas the result of evaluating the expression *x*↑3 is a number. Executing the above statement would therefore cause an error message.

We could, however, write

function *cube x*; *x*↑*3* **end;**
tab(*cube*, *1*, *0. 5*, *10*);

and this would work correctly but it is simpler to write

tab(**lambda** *x*; *x*↑*3* **end,** *1*, *0. 5*, *10*);

and the effect is identical except that no *cube* function remains after execution of the statement.

EXERCISES

1. How would you use *tab* to tabulate the values of the expression $x^2 - 2x - 1$ for integers from 0 to 100.

2. What is the value of x after execution of the following?

```
vars x k g;
lambda x; x * x end -> k;
lambda f; f(2) end -> g;
g(k) -> x;
```

12. RECURSION

A function may call itself during its execution. The POP-2 system automatically provides a distinct set of local variables when this happens.

Consider two possible ways of defining a function for computing the factorial of a number; first, an *iterative* definition using a goto statement

```
function fact n => p;
1 -> p;
loop: if n > 1 then p * n -> p; n - 1 -> n;
goto loop close
end
```

second, a *recursive* definition in which the function itself is called from within

```
function fact n;
if n > 1 then n * fact(n - 1) else 1 close
end
```

An obvious difference between these two definitions is that the second, recursive definition is much simpler to write. However, a more important difference is that execution of the recursively-defined function involves much more storage space. In fact, the whole arithmetic expression

$n * n{-}1 * n{-}2 * \ldots\ldots * 2 * 1$

is set up before it is evaluated, whereas in the first case, the result is accumulated factor by factor.

Where a choice exists between an iterative and a recursive definition, the former is usually preferable on grounds of efficiency. Often, however, the recursive definition will be briefer and more perspicuous, particularly in handling complex data structures.

As an example of the use of recursive functions in list processing let us define a function to produce the list of all items on a given list which are greater than 100. We use x for an item and xl (i.e., x-list) for a list.

```
function gr100 xl; vars x;
    if null (xl) then nil
        else hd (xl) -> x;
            if x > 100 then x :: gr100(tl(xl))
                       else gr100(tl(xl))
            close
    close
end;
gr100([90 101 85 106 107]) =>
** [101 106 107]
```

More generally if p is any property, that is, a function producing a truth value

```
function sublist xl p;vars x;
    if null(xl) then nil
        else hd(xl) -> x;
            if p(x) then x::sublist(tl(xl), p)
                    else sublist(tl(xl), p)
            close
  close
end;

function big x;x > 100
end;
sublist([90 101 85 106 107], big) =>
** [101 106 107]
```

Since we often want both the head and tail of a list the function *dest* is provided. It produces both the head and the tail. Thus we may write

```
function sublist xl p;vars x;
    if null(xl) then nil
        else dest(xl) -> xl -> x;
            if p(x) then x::sublist(xl, p)
                    else sublist(xl, p)
            close
    close
end;
```

Another example is a function to test whether an item occurs in a list.

```
function member xl xl;vars x;
    if null(xl) then false
        else dest(xl) -> xl -> x;
            if x = xl or member(xl, xl) then true else false close
        close
end;
```

EXERCISES

1. Write a recursive definition of the function *hcf* to determine the highest common factor of two integers. Also write an iterative definition of the same function. Which function is more efficient
(a) in terms of storage requirement
(b) in terms of running time?

2. Define the function *maplist* recursively (it was defined with a loop in the section on lists). You had better give it another name such as *maplist2* since *maplist* is standard.

3. What is the output of the following program?

```
function itlist xl y g;
    if null(xl) then y
        else g(hd(xl), itlist(tl(xl), y, g))
    close
end;
function add x y;x+y end;
itlist([1 2 3 4], 0, add) =>
itlist([1 2 3 4], nil, append) =>
```

4. Write recursive functions for exercises 1 and 2 of section 10 (p. 23).

13. DEFINING NEW OPERATIONS

Having defined a function such as *sumsq* with the function definition

function *sumsq x y;*
x↑2 + y↑2
end

we can evaluate expressions involving the function such as

*3 + sumsq(sumsq(4, 5 *2), 3) =>*

It is often convenient, however, to use an operation rather than an ordinary identifier to denote a function. This is standard practice in the case of arithmetic operations where it is much simpler to write

a + b + c + d

than to write

add(add(add(a, b), c), d)

where *add* is a function for adding a pair of integers. The only difference between an ordinary identifier denoting a function and an operation is that the latter has a *precedence* which can affect the order of evaluation of the expression, and hence it may be written between its agreements without any parentheses.

We can declare new operations called, say, ++ with precedence 5 and ** with precedence 3 by executing the declaration

vars operation *5* ++ **operation** *3* **;

and write 6 ++ 8 ** 10, meaning 6 ++ (8 ** 10). It is usual, though not necessary, to use identifiers made up of signs rather than letters and digits when naming operations. This convention helps the (human) reader parse an expression.

Having declared an operation, an assignment is used to assign a function value to it. It is not possible to write

sumsq —> ++

because we do not wish to perform the operation ++, only assign a value to it. To make an operation behave like an ordinary identifier, we place the word **nonop** before it. Thus the assignment

sumsq —> **nonop** ++

makes ++ into an operation for adding the square of the two expressions surrounding it. Alternatively we could simply write **operation** 5 ++ instead of **function sumsq** in the function definition.

Another use for **nonop** is when we wish to pass a function denoted by an operation variable to another function as a parameter. For example, if — — denotes a function of one argument, we could write

tab (**nonop** — —, *1, 1, 100*)
but not
tab (— —, *1, 1, 100*)

The latter would apply — — once before applying *tab* instead of applying it 100 times inside *tab*;

By associating a precedence with an identifier we can dispense with some parentheses in expressions containing that identifier. Another way of avoiding parentheses is to use the dot notation. Instead of writing $f(x)$ we write $x.f$, instead of $sin(cos(x)) + cos(sin(x))$ we write $x.cos.sin + x.sin.cos$. This is allowed when the argument of the

function is denoted by an identifier or a constant, or is itself a dot expression.

EXERCISE

POP-2 does not have a standard *not equals* operation whose value is *true* if the arguments are not equal and *false* otherwise. Define a suitable operation written /= with the same precedence as the = operation, i.e. 7.

14. MORE ABOUT LISTS

LISTS WHOSE ELEMENTS ARE LISTS

Lists may have other lists as their elements, for example,

[[1 2 3] 2 [1[2 3] 4]]

This has 3 elements, the first a list, the second a number, and the third a list of 3 elements, one of them itself a list. The following function will count how many numbers there are in such a list of lists, 8 in the one above.

```
function lengthll l;
    if null(l) then 0
        elseif islist(hd(l)) then lengthll(hd(l)) + lengthll(tl(l))
        else 1 + lengthll(tl(l))
end;
```

Note that the function *islist* recognizes lists.

Consider the following function to read a list from the keyboard. The standard function *itemread* reads one word or positive number from the keyboard, and *append* (*y*,*x*) appends the item *y* to the end of the list *x*.

```
function listread;
    vars x y;
    itemread() -> x;
    if x = "[" then nil -> y;
                    loop: listread() -> x;
                        if x = "]" then y
                                   else append(x, y) -> y
                                   goto loop
                        close
                else x
    close
end;
```

Note that a recursive definition is necessary here in order to allow lists to contain lists to any complexity.

Thus *listread*() –> *x*;
[*1* [*2 3*]]
has the same effect as
[*1* [*2 3*]] –> *x*;

DECORATED LIST BRACKETS

The list brackets described above provide a convenient notation for writing list structures consisting entirely of words or positive numbers.

Alternative *decorated brackets* [% and %] are provided for use when the individual elements of the list structure are obtained by evaluating expressions. Within decorated brackets, the expressions are separated from each other by commas. For example, the list

[%*x*, *3 + 4*, "*x*"%]

is a list of three items: the value of the variable *x*, 7, and the word "*x*". Decorated brackets can be used to create list structures of any complexity. They must be used to create lists with negative numbers since negative numbers are expressions. For example,

[% *–3.5*, *7.0*, *–2.6* %]

has a value the list of three numbers –3. 5, 7. 0, and –2. 6.

UPDATING LISTS

Given a list *x*, say [*a b c*], it is possible in POP-2 to use an assignment to change part of the list. By executing the assignment

"*d*" –> *hd*(*x*)

the list *x* becomes [*d b c*].
The function *tl* can also appear on the right-hand side of an assignment. The statement

tl(*tl*(*x*)) –> *tl*(*x*)

results in the middle item of the list being deleted and *x* becomes [*d c*]. Also, if *x* is [*a b c*], the statement

d –> *hd*(*tl*(*x*))

gives *x* the value [*a d c*].

Functions like *hd* and *tl* which can be used on either side of an assignment are called *doublets*. They actually consist of two functions, one of which is chosen for use depending upon which side of the assignment the function is called. There is a difference between the two functions of a doublet because on the left-hand side of an assignment the function must produce a value, but on the right-hand side a value must be used to change some structure. These two component functions of a doublet are called the 'selector' and the 'updater' respectively. Thus the function *sqrt* is not a doublet because there is no reasonable interpretation of the assignment

3 –> *sqrt*(*2*);

The POP-2 user may define doublets. Consider, for example, a function *element* to get the *n*th element from a list *x*. The definition of *element* is

function *element n x;*
if *n = 1* **then** *hd*(*x*) **else**
element(*n–1*, *tl*(*x*)) **close**
end;

Thus if *y* is the list [*a b c*] then *element* (*2*, *y*) is *b*. Because *element* has not been defined as a doublet, we cannot write

"*z*" –> *element* (*2*, *y*);

even though there is a very reasonable interpretation of such an assignment. That is, to replace the second element of the list *y* with the word *z*. In order to make *element* into a doublet with this meaning

when used on the right-hand side of an assignment, an auxiliary function, say *changeelement* must be defined. A suitable function is

```
function changeelement a n x;
if n = 1 then a —> hd(x) else
changeelement(a, n—1, tl(x)) close
end;
```

Notice that *changeelement* has an extra formal parameter *a* before the other two formal parameters *n* and *x* which were used in the definition of *element*. The extra formal parameter represents the value to be assigned. Having defined *changeelement*, it can itself be used to update a list. The statement

changeelement("*z*", 2, *x*)

replaces the second item of the list *x* with the word *z*; the effect required of *element* on the right-hand side of an assignment. To make *element* into a doublet, the assignment

changeelement —> *updater* (*element*)

is executed using a standard function *updater*. This assignment puts the function *changeelement* in the place that *element* goes to when called on the right-hand side of an assignment. Now we can execute the statement

"*z*" —> *element* (2, *x*)

Oddly enough, the standard function *updater* is itself a doublet. It acts on functions and can be used to get at their update part. Thus *updater* (*element*) = *changeelement* would now be *true*.

An important point to note about doublets, is that the updater function of a doublet is called only if the function is the main function on the right-hand side of an assignment. Thus in the assignment

hd(*tl*(*x*)) —> *hd*(*tl*(*y*))

which replaces the second item of list *y* with the second item of list *x*, only the function *hd* on the right-hand side uses its updater function rather than its selector. The function *tl* on the right-hand side is used in its normal sense.

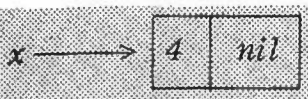
STATIC AND DYNAMIC LISTS

No mention has so far been made of the structure of lists as they appear in the memory of the machine. Lists have two representations, *static* and *dynamic*. The functions which operate on lists described so far work equally well with either static or dynamic lists, or even combinations of the two types.

The list brackets (both plain and decorated) and the *cons* operator :: all generate static lists.

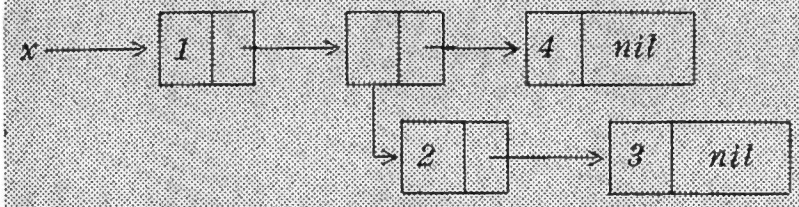
An element of a static list is called a *pair*. A pair contains two values. One is the head and the other the tail. The pair is normally represented in the computer by two adjacent memory cells and to designate a specific pair it suffices to pass the 'address' or serial number in memory of the first of these cells. Thus if we write *4*::*nil*—> *x* two adjacent memory cells are reserved and *4* and *nil* are placed in them. The address of the first of these cells is placed on the stack and then removed and placed as the value of *x*.

We may represent the situation thus, using an arrow to show that *x* contains the address of a pair.



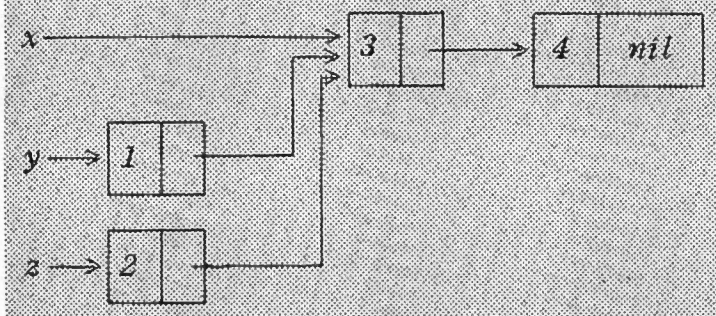
If a list x has 2 items, say, [4 6], 2 pairs are needed to represent it, thus

The second cell of the first pair contains the address of the second pair. Similarly [1 [2 3] 4] —> x gives rise to

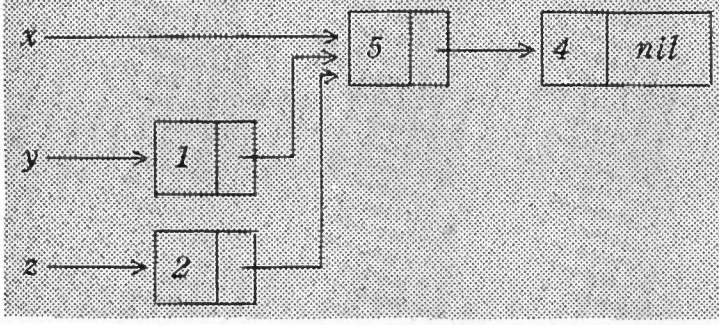


Two lists can *share* a common tail thus

```
[3 4] --> x;
1 ::x --> y;
2 ::x --> z;
```




```
5 --> hd(x);
```



We see that an assignment which updates a component of a list which is the value of one variable can alter the value of another variable.

Consider now the following piece of program

```
x =>
**[1 2 3]
y =>
**[1 2 3]
4 --> hd(y);
x =>
```

What will be printed, [1 2 3] or [4 2 3]? This depends on whether x and y share the same list [1 2 3], or have as their values distinct lists which happen to have the same elements, or indeed some intermediate situation such as sharing only the last two elements. We could find out by looking back over the preceding program, or more directly we can print $x=y$. This will be true just if x and y share the same list, not

just copies with the same elements. Similarly $tl(x) = tl(y)$ tests whether they share their last two elements.

How can we test the weaker proposition that the lists x and y have the same elements?

```
function equallist x y;
    if x = y or null(x) and null(y) then true
        elseif null(x) or null(y) then false
        elseif hd(x) = hd(y) and equallist(tl(x), tl(y)) then true
        else false
    close
end;
```

Thus [1 2 3] = [1 2 3] is *false* but *equallist* ([1 2 3], [1 2 3]) is *true*.

Equallist as defined assumes that the elements of the given lists are to be tested for strict equality, not merely list equality, and *equallist* ([1[2 3]], [1[2 3]]) is false. To make *equallist* test for element-wise equality throughout put **if** $atom(x)$ **or** $atom(y)$ **then** $x = y$ **exit**; before **if** and to replace $hd(x)=hd(y)$ by $equallist(hd(x), hd(y))$.

We will see later that the *pair* used to build static lists is just a special standard kind of *record*.

If the elements of a list are arbitrary and have no relation to each other, the static representation is very suitable. If, however, each element of a list is related to its predecessor by a well-defined rule, the list can be represented by this rule rather than by the actual elements. Thus the rule 'add 1' could be used to represent the infinite list 0, 1, 2..... A dynamic list is a list represented by a rule in the form of a POP-2 function. The function must be a function of zero parameters and must always yield exactly one result. The function must be so written that each successive call generates the successive elements of the list. There is a built-in function called *fntolist* which converts such a function into a dynamic list.

Consider how the dynamic list, 0, 1, 2, and so on, might be constructed. The following will achieve it.

```
vars n;-1 -> n;
function suc;n + 1 -> n;n end;
fntolist(suc) -> y;
```

The above POP-2 text produces the dynamic list y, which behaves just like any static list except that very little storage space is required. For example, the function *element* defined above to extract the nth item from a list will work just as well with a dynamic list as with a static list. Thus

```
element(y, 20) =>
```

produces the output

```
**19
```

because the twentieth item on the list y is the integer 19.

An important use of dynamic lists is to represent a stream of items read from an input device. For example, the function *itemread* described earlier, which reads one item from the keyboard, can be turned into a dynamic list,

```
fntolist(itemread) -> x
```

enabling any program that processes a list of items to work on items typed directly on the keyboard.

Dynamic lists provide a variant of the facility called a *stream* devised by Landin (1965).

EXERCISES

1. The function *makeassoc* was previously defined as

```
function makeassoc x y xyl => xyl1
    x::(y::xyl) --> xyl1
end;
```

Rewrite it so that if x is already on the list xyl it changes the associated value to y producing the altered list as a result.

2. What is the output of the following program?

```
[1 2] --> x;
x --> x.tl.tl;
x.hd,x.tl.hd,x.tl.tl.hd,x.tl.tl.tl.hd =>
```

3. Define a function *edit* which has as parameters three lists. The function should look for the second list within the first list and replace it with the third list. For example,

```
[jim is a son of a bitch and so is bob] --> xl;
edit (xl, [son of a bitch], [* * * *]) --> xl;
xl =>
** [jim is a * * * * and so is bob]
```

4. Write a function to produce as a dynamic list the prime numbers from 1 to n.

15. RECORDS

The pair described in the previous section is a special case of a *record*. A pair consists of two components called the *front* and the *back*. When a pair is used as an element in a static list, the functions *hd* and *tl* refer to the front and back of the pair respectively. A pair is created by the function *conspair*, which finds an area of memory and places two values in the front and back of the new pair. The function *cons* used in list processing is the same as *conspair*. *Conspair* is called the *constructor* for pair records. Just as a constructor takes the components of a record and produces a record containing the components, there is a complementary function, called a *destructor*, which takes a record and yields the components of the record as results. In the case of a pair, the destructor function is *destpair*, which takes a pair and produces the front and back components of the pair as results. Note, however, that in spite of its name, a destructor does not actually destroy the record; it merely extracts its components.

The pair can be used by the POP-2 programmer in many ways. It is not restricted to its use in list processing. A pair could be used to represent a complex number and functions and operators defined for handling pairs representing complex numbers. Thus 1::2 represents the number $1 + 2i$. The following might serve as a basis for complex number manipulation.

```
operation 6 +++ x y;
conspair(front(x) + front(y), back(x) + back(y))
end;
```

```
operation 6 --- x y;
conspair(front(x) - front(y), back(x) - back(y))
end;
operation 5 *** x y;
conspair(front(x) * front(y) - back(x) * back(y)
front(x) * back(y) + back(x) * front(y))
end;
operation 5 /// x y;
vars z; sqrt(front(y)↑2 + back(y)↑2) -> z;
conspair((front(x) * front(y) + back(x) * back(y))/z,
         (back(x) * front(y) - front(x) * back(y))/z)
end;
(-1)::2 -> u;
1::2 +++ 3::(-3) *** u -> v;
v.front, v.back =>
** 4, 11
```

The record therefore enables a collection of quantities to be known by one name. This is useful not only for complex numbers but for a wide variety of situations, such as constructing list processing functions for lists with both forward and backward pointers or storing several items of information about an individual employee. The pair record cannot be used of course if more than two items of information are associated with the particular object.

POP-2 provides facilities for defining new records and functions for dealing with them. If the pair was not already defined it could be defined using the *recordfns* function

```
recordfns ("pair",[0 0]) -> back -> front -> destpair -> conspair;
```

The standard function *recordfns* takes two parameters: the name to be associated with the type of record being defined, and a list of integers. The number of integers in the list indicates the number of components the records are to have—in this case two. In this list, the integer zero indicates space for storing a POP-2 value just like any variable. Any value other than zero indicates the number of bits required to store the particular component. This enables more than one component to be stored in a single machine word. Instead of an integer we may have "*COMPND*", meaning the component must be a compound item, i.e., not a real or integer.

It is important to note that *recordfns* does not produce records; it produces functions for handling a new class of records. In the case of the pair, *recordfns* places on the stack the constructor function *conspair*, the destructor function *destpair*, the doublets for accessing the two components *front* and *back* (called select/update doublets because they allow us either to select out part of a record or to update that part of the record, giving it a new value). In order to use these functions they must be taken off the stack and assigned to variables. Note that with functions producing more than one result, the order of assignment is the reverse of the order in which results are placed on the stack.

Consider the problem of handling information about a collection of persons. Each person can be represented by a record with three components indicating the person's name, age, and sex. The name can be represented by a word, the age by an integer less than 128, and the sex by the integers 0 or 1. We can therefore set up functions for handling such records as follows:

```
recordfns ("person", [0 7 1]) -> male
-> age -> name -> destper -> consper;
```

This defines and names five new functions, and the following shows how they might be used. First *consper* can be used to construct a few records

```
consper("smith", 31, 1) -> p1;
consper("jones", 21, 0) -> p2;
consper("robinson", 93, 1) -> p3;
```

*p*1, *p*2, and *p*3 are now records of type *person*, and can be interrogated or updated by the doublets *name*, *age*, and *sex*.

```
name(p2) =>
**jones
function birthday p;
age (p) + 1 -> age (p)
end;
birthday(p3);
age(p3) =>
**94
function marry boy girl;
if male(boy) and not(male(girl)) then
name(boy) -> name(girl) else pr("shame") close
end;
marry(p1, p3);
shame
marry(p1, p2);
name(p2) =>
**smith
```

Sometimes we wish to test a record to see to which class it belongs. The standard function *dataword* produces the word associated with the class.

Thus

```
dataword(p1) =>
**person
```

The record facility of POP-2 permits the user to define new compound objects out of existing objects, thus extending the language to handle quantities associated with a class of problems. The objects could be represented instead by list structures, but the appropriate record structures usually take less storage space and the use of specially-named functions (select-update doublets) to access the components makes programming easier and clearer.

EXERCISES

1. A point can be represented by two real numbers. A triangle can be represented by three points. Define classes of records to represent points and triangles. Define a function *equilateral* which tests if a given triangle has sides of equal length.

2. A flight has a number, a starting place, a finishing place, a starting time, and a finishing time. Given a list of flights, write a function to get to a given place by a given time starting from a given place at a given time.

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).

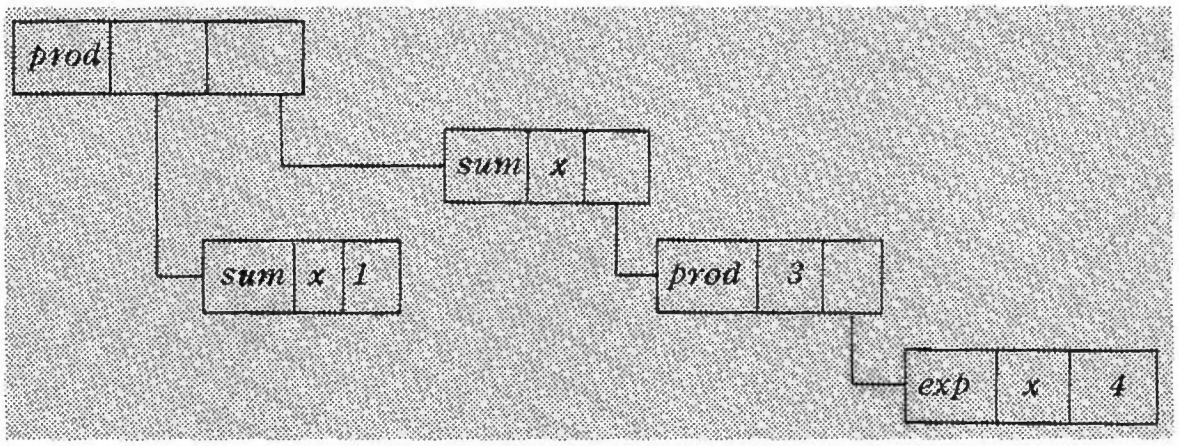
```
vars sum1 sum2 destsum operation 4 ++;
recordfns ("sum", [0 0]) -> sum1 -> sum2 -> destsum -> nonop ++;
vars prod1 prod2 destprod operation 3 **;
recordfns ("prod", [0 0]) -> prod1 -> prod2 -> destprod -> nonop **;
vars exp1 exp2 destexp operation 2 ↑↑;
recordfns ("exp", [0 0]) -> exp1 -> exp2 -> 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 $x\uparrow\uparrow x$, 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. S T R I P S

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 *n*th 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 *n*1 places before the decimal point and *n*2 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. J U M P O U T

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.                                        ** DB

: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 A
          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<6 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 SQRTREP 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;
```