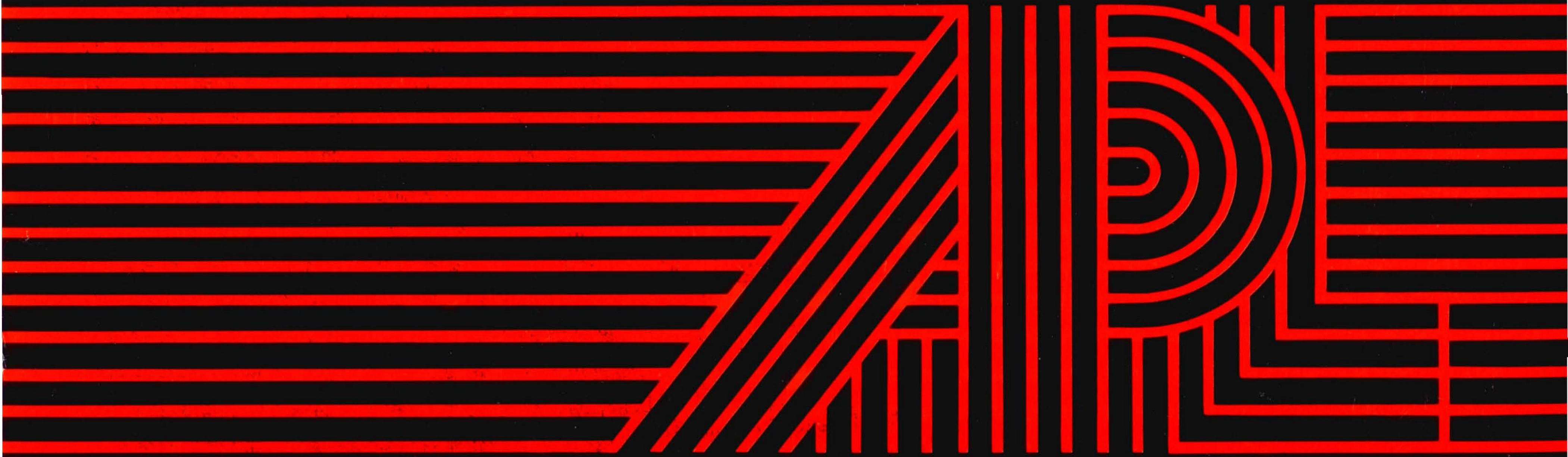
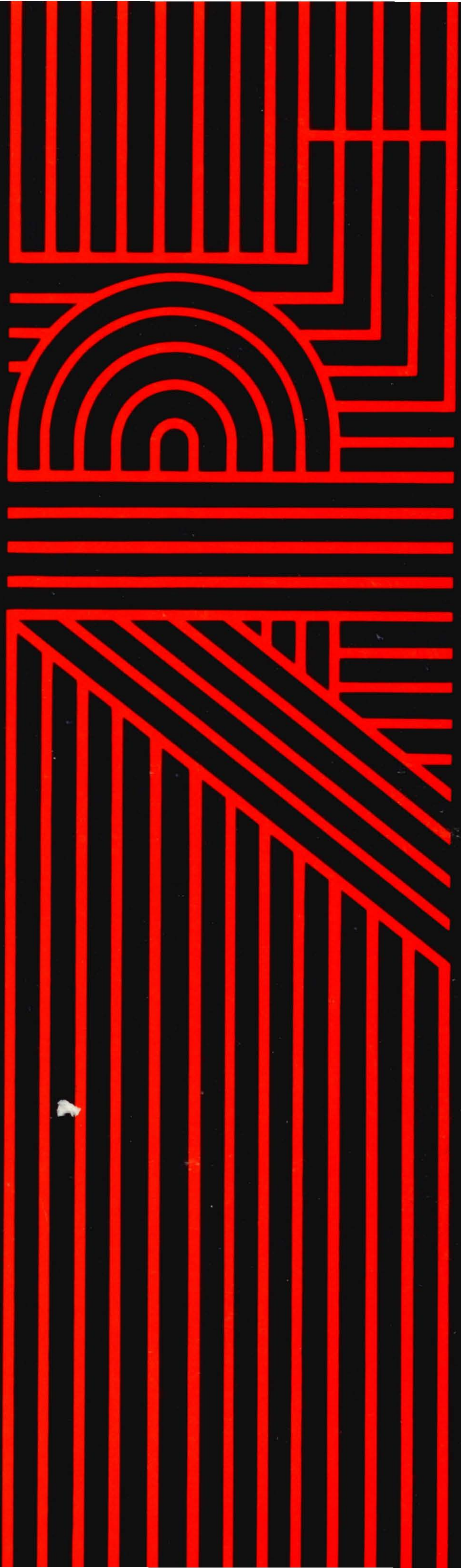
Kenneth E. Iverson

A Working Introduction to APL

From the Education Group
I.P. Sharp Associates



I.P. Sharp Associates



PREFACE

Preface

I have long been struck by the contrast between the success with which the adventurous learn APL by simply using it, and the frequent failure of lecture courses to communicate the simplicity and applicability of the language. I have also felt that it should be possible to incorporate into a formal course the advantages of learning by exploration with an APL terminal, and I therefore welcomed the opportunity to begin the design and testing of such a course in early 1980.

Because of the nature of our business as a vendor of APL service for commercial users, we have aimed this course at people already active in some professional capacity, particularly those who have recognized the potential of computers in their work as well as the frustrations commonly associated with conventional computer languages and systems. For such people we proposed to provide in a three-day course "the ability to translate into APL, and therefore into computer use, procedures of interest in your own profession", and to provide the familiarity with reference material and with techniques of experimentation which are essential to independent work and to further growth in the mastery of the language.

The course has undergone many changes in its development, evolving to give a student full freedom in choosing his own pace, to accommodate a wider range of student backgrounds and interests, and to apply to APL systems other than our own. Nevertheless, this final version has been used by hundreds of students, and the concomitant course for instructors has been used repeatedly.

I am indebted to a great number of my colleagues at I.P. Sharp Associates. On the technical side, I am pleased to acknowledge the work of Roland Pesch in designing the necessary file functions, and in participating in the design and testing of the first version of the course. On the teaching side, I have benefited from suggestions from a number of instructors who used the course, particularly Nancy Petersons, Paul Berry, and R.C. Metzger.

On the administrative side, I have enjoyed the essential support of Lael Kirk and Rosanne Wild in designing and producing the final form of the work. On the clerical side, I have benefited from the competent and cheerful services of Ginger Kahn and Deborah Rodbourne.

Finally, I must acknowledge the essential contribution of those strong-willed colleagues steeped in conventional lecture methods who forced me to thoroughly think through the preparation needed for instructors of a course of this nature.

Kenneth E. Iverson
Toronto, Ontario
July, 1981

INSTRUCTOR'S
HANDBOOK

A Working Introduction to APL is a course designed to develop independence and self-reliance by fostering a faith in the ability to learn by experimentation and by the use of reference material. The tutoring provided is therefore crucial: an instructor must not destroy this faith by plunging the student too quickly into complexities, by putting him off with the remark that a matter is too difficult to discuss at the moment, or by providing lengthy lectures that relieve the student of the burden of exploration.

The instructor is not required to have previous experience in lecturing, but is expected to be already competent in APL. The present course for instructors is therefore aimed at providing: 1) familiarity with the student material, 2) experience in handling student's questions, 3) experience in presenting brief summary lectures, and 4) a deeper knowledge of the structure and rationale of APL than that normally gained in practical application of it.

The major emphasis in this course for instructors is placed upon the handling of student questions. It proceeds in the following stages:

1. The introductory lecture is presented (on slides) as it will be to students.

2. Instructors work through the first four sessions of the student material. In this they should simulate students as much as possible, except that (already knowing APL) they should require only about one hour, and that they should be considering how they would handle relevant questions from *their* students.

3. Instructors will be exercised orally in the handling of a series of student questions relevant to the sessions covered.

4. Instructors will be given twenty minutes to prepare a suitable brief summary of the material covered, and one or more will be asked to present his summary for discussion.

5. The slides provided for the first summary lecture will be presented and discussed.

6. The **Instructor's Guide** will be reviewed and discussed.

7. Steps 2-5 will be repeated for each group of four student sessions.

8. The student's supplementary exercises will be reviewed and annotated.

9. The source material will be reviewed and annotated.

10. The facilities for remote tutoring will be used and discussed.

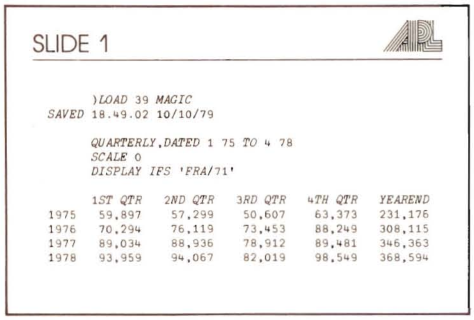
In the section devoted to lectures, the background material for the instructor is presented in a narrow format together with comments on the use of the slides; the slides themselves appear in the margin.

LECTURE I
Introduction

...they know enough who know how to learn.
Henry James

The purpose of this course is to teach how to program in APL; the purpose of this introduction is to clarify this purpose and to outline the teaching methods to be used.

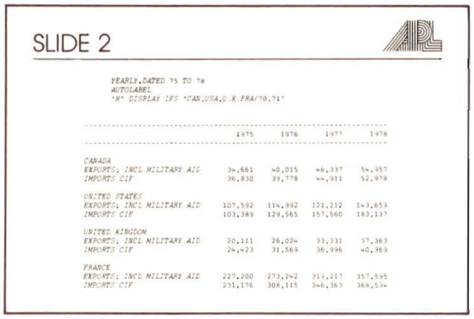
Slide 1 shows an example of the use of computers, obtained by simply following the instruction in the user's manual for a certain computer application designed for the handling of data bases The series shown represent France's imports by quarter, in billions of French francs. When MAGIC is loaded, several default settings are already set, including an option called *TOTALS* that sums each period in a year (quarters in this example) to form yearend totals.



SLIDE 1

```
       )LOAD 39 MAGIC
SAVED  18.49.02 10/10/79

       QUARTERLY,DATED 1 75 TO 4 78
       SCALE 0
       DISPLAY IFS 'FRA/71'

       1ST QTR   2ND QTR   3RD QTR   4TH QTR   YEAREND
1975    59,897    57,299    50,607    63,373   231,176
1976    70,294    76,119    73,453    88,249   308,115
1977    89,034    88,936    78,912    89,481   346,363
1978    93,959    94,067    82,019    98,549   368,594
```

Slide 2 illustrates how several series may be accessed and displayed at the same time.

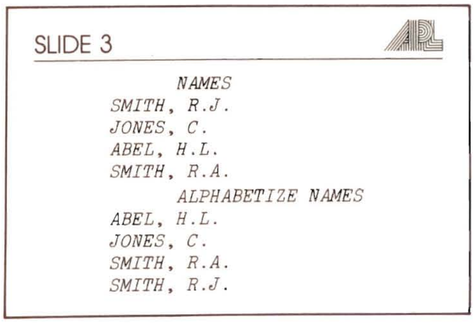


SLIDE 2

```
       YEARLY,DATED 75 TO 78
       AUTOLABEL
       'H' DISPLAY IFS 'CAN,USA,U.K.FRA/70,71'

--------------------------------------------------------------
                                  1975     1976     1977     1978
--------------------------------------------------------------

CANADA
EXPORTS; INCL MILITARY AID      34,661   40,015   46,337   54,957
IMPORTS CIF                     36,830   39,778   44,911   52,978

UNITED STATES
EXPORTS; INCL MILITARY AID     107,592  114,992  121,212  143,659
IMPORTS CIF                    103,389  129,565  157,560  183,137

UNITED KINGDOM
EXPORTS; INCL MILITARY AID      20,111   26,024   33,331   37,363
IMPORTS CIF                     24,423   31,569   36,996   40,969

FRANCE
EXPORTS; INCL MILITARY AID     227,200  273,242  319,217  357,595
IMPORTS CIF                    231,176  308,115  346,363  368,594
```

The **programs** or **functions** (such as *SCALE*, *DISPLAY*, and *IFS*) used in the expressions entered to invoke the printing of the tables in Slides 1 and 2 are themselves constructed from simpler and more general primitives in the **programming language** used, in this case APL.

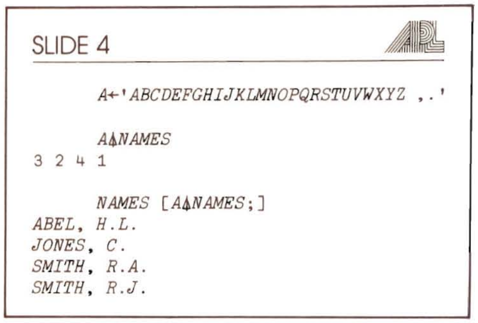
Suppose, for example, we wish to construct a function *ALPHABETIZE* which when applied to a table of names would produce the same table in alphabetical order, as illustrated in Slide 3. We could begin by specifying the ordering of the alphabet to be used, and continue by using the primitive function ⍋ (called **grade**) to determine the order in which the rows of the argument *NAMES* are to be selected, and by using that result to select the rows in desired order.



SLIDE 3

```
          NAMES
SMITH, R.J.
JONES, C.
ABEL, H.L.
SMITH, R.A.
          ALPHABETIZE NAMES
ABEL, H.L.
JONES, C.
SMITH, R.A.
SMITH, R.J.
```

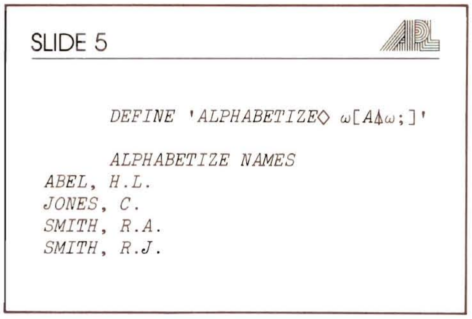
These steps are shown in Slide 4. The instructor should avoid detailed discussions (such as the precise definition of ⍋); at this point such detail will not help in communicating the overall definition of programming.



SLIDE 4

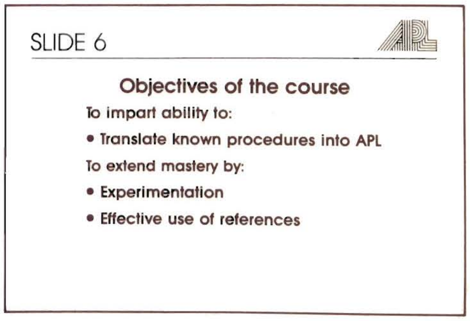
```
      A←'ABCDEFGHIJKLMNOPQRSTUVWXYZ ,.'

      A⍋NAMES
3 2 4 1

      NAMES [A⍋NAMES;]
ABEL, H.L.
JONES, C.
SMITH, R.A.
SMITH, R.J.
```

It remains to state that the name *ALPHABETIZE* is to be assigned to the sequence of operations specified by the last expression. This is done as shown in Slide 5, using the Greek letter omega to stand for the argument of the function. Slide 5 is an example of **programming**, that is, the construction of a desired function from a set of available primitive functions.



SLIDE 5

```
        DEFINE 'ALPHABETIZE◇ ω[A⍋ω;]'

        ALPHABETIZE NAMES
 ABEL, H.L.
 JONES, C.
 SMITH, R.A.
 SMITH, R.J.
```



SLIDE 6

Objectives of the course

To impart ability to:

- Translate known procedures into APL

To extend mastery by:

- Experimentation
- Effective use of references

Objectives of the course. Slide 6 may be used to clarify the objectives and should be presented in the light of the following discussion.

The overall purpose is to teach how to program in APL. The student should, however, be aware of the limitations of a brief, general course in programming:

> Because it is a course in **programming**, it concentrates on presenting the primitives of the language used, and on the methods of constructing desired functions from them, **not** on how to choose or analyze the functions important to a particular area of application. Therefore, in order to apply his programming ability effectively, a student must know or acquire knowledge of the functions and procedures of interest in some area of application. Fortunately, general non-specialized knowledge (such as filing, bookkeeping, and high school algebra), combined with imagination and a penchant for exploration, provides a sufficient basis for a good deal of programming.

> Because it is aimed at a general audience having a variety of backgrounds and interests, the examples used are not chosen to suit any one specialty such as accounting, engineering, or actuarial work. The examples are therefore rather general in nature, and it may require some imagination (and perhaps some help from the instructor) to see how various techniques might apply to the student's own field of interest. The development of such imagination is essential to good programming.

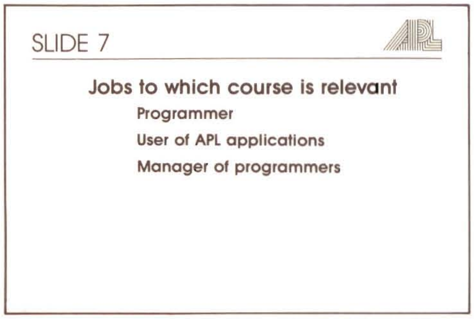
> Because it is brief, the course cannot effectively explore, or even introduce, all of the primitives available in the language. Emphasis is therefore placed on learning how to learn, that is, on learning how to resolve questions by experimentation and by effective use of reference material rather than by reliance on an instructor. In other words, the course·is intended to provide a solid basis for further growth rather than a more superficial acquaintance with all parts of the language used.

Introductory programming courses differ radically in their objectives, and students should be aware of this. Courses often aim at exposing the student to as many primitives of the language as possible. The objectives of the present course are more practical:

> To impart the ability to translate into APL, and therefore into computer use, procedures of interest in a student's own profession and known or available to him in terms understood in that profession.

To provide sufficient familiarity with APL reference material and with techniques of experimentation to allow students to continue to expand their mastery of APL through further use and independent study.

Student's Aims. Students may choose to attend an introductory programming course not only with the ultimate aim of programming, but with related objectives such as the use of prepared programs, or the management of programmers or computer resources. The following disucssion may be used to guide the use of Slides 7-10 in treating these matters:

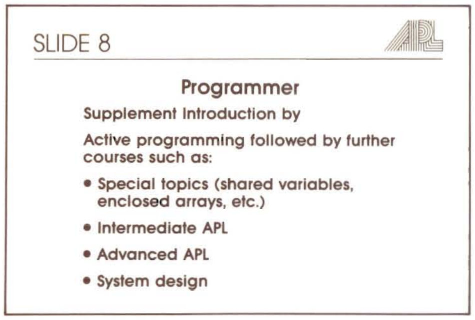

SLIDE 7

Jobs to which course is relevant

Programmer

User of APL applications

Manager of programmers

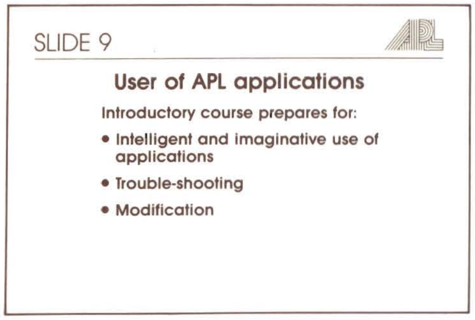

SLIDE 8

Programmer

Supplement Introduction by

Active programming followed by further courses such as:

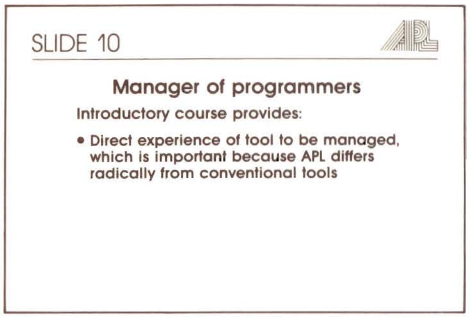
- Special topics (shared variables, enclosed arrays, etc.)
- Intermediate APL
- Advanced APL
- System design


SLIDE 9

User of APL applications

Introductory course prepares for:

- Intelligent and imaginative use of applications
- Trouble-shooting
- Modification


SLIDE 10

Manager of programmers

Introductory course provides:

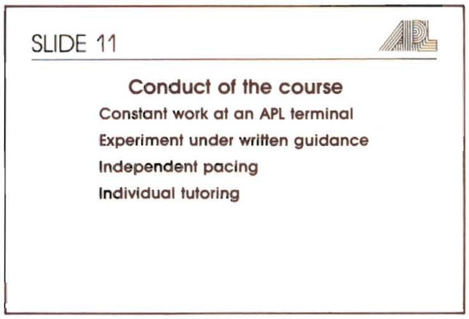
- Direct experience of tool to be managed, which is important because APL differs radically from conventional tools

Programmer. Although this course is aimed at teaching programming, the serious programmer should eventually supplement it by courses such as 1) Special Topics in APL (e.g., shared variables, formatting, files), 2) System Design in APL, or 3) Advanced APL, covering all aspects of the language. In particular, anyone planning to become an APL System Programmer (designing and producing application packages for use by others) should obtain further instruction and experience in system design and documentation.

However, a graduate of an introductory course should probably attempt to work in APL for a month or more before continuing with further courses.

User. Although a user of APL-based applications need not know how to program in APL, such knowledge can be helpful in making intelligent use of applications. In particular, the functions provided in applications can often be supplemented in significant ways by APL expressions. Moreover, even a programmer with limited experience can often define new functions employing the functions, provided in an application as he would primitives.

Manager. A manager can usually benefit from some first-hand knowledge of the tools used by his subordinates, and direct hands-on work with these tools is an effective way of gaining such knowledge. Even managers already familiar with a variety of conventional programming languages will probably find the present course helpful, since APL differs rather radically from conventional languages.



SLIDE 11

Conduct of the course

Constant work at an APL terminal

Experiment under written guidance

Independent pacing

Individual tutoring

Teaching Method. The teaching method used is pragmatic, and differs radically from conventional lecture methods:

> The method used allows you to begin work immediately on an APL terminal, experimenting under the guidance of simple written instructions, and proceeding as independently of the instructor as your ability permits. Two students work together at each terminal, to their mutual benefit, and proceed at their own pace independently of other student pairs.
>
> The instructor therefore spends little time in formal lectures, and most of the time in individual tutoring -- answering specific questions (perhaps by directing attention to points covered in earlier sessions), and suggesting general techniques for resolving questions through further experimentation and further use of the reference material.

The foregoing may be discussed with the aid of Slide 11.

LECTURE II
Sessions 1—4

Handling Questions

The main points to be made in the handling of questions are indicated in the following list:

DO:

a) Answer a simple question of fact simply and directly.

b) Where possible, answer by suggesting experiments which will allow the student to resolve the question (and possibly related questions) for himself. These experiments may involve the terminal, or may, as in questions concerning catenation of arrays, involve manual work such as the drawing of rectangles to represent the shapes of tables.

c) Use every occasion to refer the student to sources in manuals, particularly if his curiousity is outrunning the pace of others.

d) If the source of difficulty is not clear, show the student how to pinpoint it by breaking a composite expression into its constituents to find the simplest form in which trouble appears.

e) In general, treat questions so as to develop the student's ability to analyze difficulties for himself and to use reference material effectively.

f) Encourage students to display and examine results (such as *LISTOFTABLES* in Session 2) produced.

DO NOT:

a) Do not use a simple question as an excuse to launch into a general discussion of some notion which the student may have just encountered and whose ramifications he is as yet incapable of grasping. For example, if he is able to use expressions such as +/ and ×/ correctly, do not immediately burden him with a general discussion of operators just because / happens to be one.

b) Do not use a simple question as an excuse to cross-examine the student on your own pet obsessions with language difficulties; wait until he encounters the difficulty himself. If he has absorbed the methods of experimentation he should be able to analyze such problems for himself.

c) Do not annoy the student by carrying the insistence on consulting manuals to excess. For example, if a student asks the name of a particular symbol, tell him, unless you suspect that he has not yet learned about the relevant table in the manual.

d) Do not continue discussion if the difficulty has been identified as something already treated in an earlier session. Instead, give the student the opportunity to review the earlier session and then reopen the discussion if he still requires help.

Sample dialogue. The answers provided should be a sufficient guide to the teacher in conducting dialogues based on the exercises. However, we will first give a sample dialogue for Session 1, denoting the student by S, and the instructor by I.

S: I just entered 3⋆*X*÷2 and got 9, which I find confusing.

I: What value was assigned to *X*?

S: Here is the whole thing:

```
      Y←2
      X←Y+2
      X
4
      Y←3
      Y+2
5
      3*X÷2
9
```

I: Yes, you have misunderstood the order of execution. You see, in APL there is no hierarchy, and the * is not necessarily done before the ÷.

S: (Sotto voce) What in the world is hierarchy? [This term has not occurred in sessions.]

I: Furthermore, APL has the odd rule that you execute expressions from right-to-left, so that you must perform the *X*÷2 first.

S: I know that the *X*÷2 is done first, because a friend told me that about APL yesterday. But it still doesn't come out right. Oh, I guess what I didn't understand was that "right-to-left" implies that *X*÷2 means 2 divided by *X*.

I: No, no, forget that. Now I see the problem. You have misunderstood the meaning of specification, and the value of *X* is not what you thought.

S: (Sotto voce) What in the world is specification?

I: You see, when you entered *X*←*Y*+2, the *Y* was evaluated at that time, giving *X* the value 4. Consequently, your later respecification of *Y* had no effect on *X* even though the expression *Y*+2 used to specify *X* now means 5. Therefore, the *X* in your expression 3**X*÷2 has the value 4.

S: I know that. In fact, I printed out its value just above.

I: So you did. But what is the problem then?

S: Well, *X*÷2 gives 2, but 3*2 should give 6, not 9. Are the results in APL expressed in some weird number system?

I: No, no, they are in decimal. But why should 3*2 be 6?

S: Because 3 times 2 is 6.

I: Hmmm!

I: BUT DIDN'T YOU SEE × USED FOR MULTIPLICATION AT THE VERY BEGINNING OF SESSION 1?!

S: Yes, but I know that * also means multiplication on computers.

I: Not in APL. You will find its use in the middle of the page.

S: I hadn't gotten there yet. Oh yes, **power**! Of course, 3 to the power 2 is 9.

S: My friend was right about the absurd difficulty of APL. This is going to be a long three days!

The foregoing dialogue may seem extreme, but worse can happen. Such exchanges not only waste precious time, but they destroy that belief in the simplicity and orderliness of the topic under investigation which must underlie any serious attempt to learn by experimentation.

An attempt to help the student pinpoint the difficulty normally proves more effective. For example:

S: I just entered `3⋆X÷2` and got `9`, which I find confusing.

I: Even such a simple expression conprises successive steps. At what step does it seem to go wrong?

S: Well, I just displayed `X` to see that its value is `4`, and `X÷2` gives `2` correctly, but `3⋆2` should be `6`, not `9`.

To develop an instructor's ability to handle questions from students, the exercises below should be used as follows:

a) The teacher (simulating a student) poses a question from the list to one of the instructors, and then continues a dialogue as a student might, often leading the instructor into a quagmire as in the foregoing sample dialogue.

b) Other instructors are invited to comment on the handling of the question.

c) The corresponding answer is read out, and discussion by the instructors is invited.

To make this dialogue as effective as possible, it is essential that **neither the exercises nor the answers be given to the instructors in advance**. Moreover, instructors should be forced to simulate real dialogue; do not allow an instructor to say "in answer to such a question I would say so-and-so"—force him to simply say so-and-so.

1.1 Why doesn't `9-3-2` give `4`?

____

Because the result depends upon the order in which the parts are executed, this could be either `(9-3)-2` or `9-(2-3)`. If you want to be explicit you should use parentheses.

The order in which the parts of an unparenthesized expression are executed will be treated in Session 6. You now have several options:

1. Until you reach Session 6, use complete parenthesization to specify the order you want.

2. Experiment with expressions like `2×(3+6)` and `(2×3)+6`, and `2×3+6` to determine what the rule for unparenthesized expressions is.

3. Consult the IBM manual either to determine the rule or to confirm the conclusions you reached in option 2.

I would recommend Option 1, because your time might best be spent in continuing with the work outlined in the sessions.

1.2 I entered *X*←3 and *Y*←4. Why won't *XY* (or *X Y*) give me the product 12?

____

In any notation one may be able to omit the name of **one** of the functions used without introducing ambiguity. In mathematics one is allowed to omit the symbol for **times**, as in *XY*+*Z* instead of *X*×*Y*+*Z*. However, the possibility of such omission has certain unpleasant consequences. For example, *AREA* then means *A*×*R*×*E*×*A* and cannot be used as a (mnemonically attractive) name instead of a single-letter name such as *A* or *X*.

1.3 What is the meaning of *DOMAIN ERROR*? I entered 4÷0.

____

let's look up the message in the ref. manual

It means that the expression 4÷0 cannot be meaningfully executed, that is, it cannot yield a result which if multiplied by 0 would yield 4. In other words, the pair of arguments 4 and 0 is not in the **domain** (i.e., "territory or range of rule or control" [American Heritage Dictionary] of the division function.

Be sure to confirm this by reading the Table of Error Messages when you receive your APL manuals at the end of Session 2.

1.4 Why did I get *LENGTH ERROR* when I entered 3 4 5 + 6 7 8 9?

____

0 is identity element for addition, and 1 is ok for multiplication, but what about ⌈?

⌈/⍳0 ⇒ identity element

Because the two lists 3 4 5 and 6 7 8 9 are to be added element-by-element, and they do not have the same number of elements, that is, the same **length**.

Be sure to consult the Table of Error Messages in the APL manual when available.

If you think that 3 4 5 + 6 7 8 9 should yield the same as 3 4 5 0 + 6 7 8 9, then consider whether it might rather be 0 3 4 5 + 6 7 8 9, and also whether this extension would be useful in conjunction with other functions such as × and ⌈.

1.5 Why does three-quarters times one-half give three eights (or *DOMAIN ERROR* on systems which do not have **replicate**) rather than three-eighths? I entered 3/4 × 1/2.

____

Division is denoted by ÷, **not** by the slash (/). Consequently, you might write (3÷4)×(1÷2), or more simply, .75×.5.

The slash denotes an entirely different function which you could try to explore by experimentation or by consulting the manual, but such a digression is probably unwise at this point.

1.6 What do you call the symbols ⌈ and ⌊?

____

floor or minimum

The **symbol** ⌊ is called "down-stile"; the function it represents in an expression such as ⌊3.14 is called "integer part", or "floor".

ceiling or maximum

The symbol ⌈ is called "up-stile",; the function it represents in the expression ⌈3.14 is called "ceiling".

APL manuals give tables of these symbols together with their names. Consult them when available.

1.7 I forgot to put a 3 in front of ⋆2 and got a weird result. What does it mean?

Just as the minus sign in the expression -*Y* represents a different function (negation) than it does in *X*-*Y* (subtraction), so ⋆ in the expression ⋆2 represents a different function (exponentiation) than it does in 3⋆2 (power).

If you know about exponentiation and the number *e* (approximately 2.71828) then you might try a few experiments such as *E*←⋆1, and ⋆*X* and *E*⋆*X* for various values of *X*. If you are not familiar with such matters, forget it.

1.8 A friend told me that ⌊ means the minimum function, but here you say it means to round down!

as shown in Berry, p.2B, or IBM p32, most functions can be used w/ either 1 or 2 arguments.

Yes, when used with two arguments, as in 2.718 ⌊ 3.14, the "down-stile" symbol does denote the minimum function, and yields the lesser of its arguments; in this example, 2.718. When used with one argument, the down-stile means "floor" or "integer-part", and ⌊3.14 yields 3. This double use of the symbol ⌊ parallels the double use of the minus sign (-) in mathematics to mean both subtraction (*A*-*B*) and negation (-*B*).

2.1 A friend told me that ⍳3 means 0 1 2, but here it seems to mean 1 2 3?.

The integer with which the sequence produced by ⍳3 begins is determined by the value of the special variable ⎕*IO*. It may properly be set to either 1 (by ⎕*IO*←1) as it now is, or to 0 (by ⎕*IO*←0) as you may wish to set it to experiment with it. You may also want to read about "Index origin" in the manuals.

However, if you change ⎕*IO* be sure to reset it to 1, since succeeding sessions depend on this setting.

2.2 Would it be possible to do *TABLE*⋆*TABLE* as well as *TABLE* × *TABLE*?

Try it, but first sketch out in pencil what you think the result might be.

2.3 Why does ⍴3 fail to print a result?

The result of ⍴3 is an **empty vector** whose length (i.e., number of elements) is zero. The ramifications of this point may be more than you should attempt to explore completely at this point. However, the following experiments can be performed fairly quickly and should give a good deal of insight into the matter:

a) The expression ⍳*N* gives a vector of *N* elements. Experiment with the expressions ⍳*N* and ⍴⍳*N* for various values of *N* including 0.

b) Repeat the experiments ⍴*LIST* and ⍴*TABLE* and ⍴*LISTOFTABLES*, and note that the number of elements in each result is the number of indices required to select a single element from the original argument. Thus, the selection of an element from *TABLE* requires the specification of two indices (the row number and the column number), and ⍴*TABLE* has two elements which specify the ranges of each of these indices.

A single quantity such as 3 (called a **scalar**) requires no index to select its only element, and ρ3 therefore has no elements.

c) The expression ρρ*X* yields the number of elements in ρ*X* and is called the **rank** of *X*. Experiment with the expression ρρ*X* for various arguments including the scalar 3 and those used in part b.

d) Make a note to consult your manuals on this matter when you receive them. Use the index for references to "rank" or "rank of an array".

2.4 The term "vector" you used in discussing the last question is new, but seems to be synonymous with "list". Is this so?

————————————

Yes, I used the term "vector" because that is what you will probably find used in the manual and in other APL literature.

2.5 Since *TABLE* is 2 3ρ*LIST*, I should be able to get *LIST* back from it. How can this be done?

————————————

Yes, 6ρ*TABLE* yields the same as *LIST*. You might also examine the effect of (×/ρ*X*)ρ*X* applied to any argument *X*. Make a note to compare the foregoing expressions with the use of the **ravel** function, either by consulting a manual or by observing its use in Session 4.

2.6 Why would anyone use something like *LISTOFTABLES*?

————————————

In Session 6 you will see the tables *OIL*72 and *OIL*73 which give oil imports (by Country by Quarter) for the years 1972 and 1973, and a "list of tables". called *OIL* which is arranged by Years by Country by Quarter. Uses of these are explored in later sessions, showing summation over years, summation over countries, maxima over years, etc.

2.7 Is it possible to make a table of tables?

————————————

Try 2 3 4 5ρι120 and 2 3 4 5ρ9 and (after trying ?9 **several times**) ? 2 3 4 5ρ9.

3.1 What is the meaning of the first symbol in ⌈/*LIST*?

————————————

This is your first opportunity to use the tables (in this case Figure 6) which you marked in your manual at the end of Session 2.

3.2 Why would one want to do things such as +/[1]*TABLE* and +/[2] *TABLE*?

————————————

If *TABLE* were a table of oil imports for some year arranged by Country by Quarter (such as the tables *OIL*72 and *OIL*73 which you will encounter in Session 6), then +/[1]*TABLE* would yield four (quarterly) sums over all supplying countries, and +/[2]*TABLE* would yield a number of (country) sums for the year.

3.3 Why do +/*TABLE* and +/*LISTOFTABLES* work without using numbers in brackets?

____

Compare the results of +/*TABLE* with +/[*K*]*TABLE* for different values of *K* (including +/[1]*TABLE* and +/[2]*TABLE*) and determine which is the "default" case of *K* provided by the simple case +/*TABLE*. Repeat for the variable *LISTOFTABLES*.

3.4 Aren't the results of +/[1]*TABLE* and +/[2]*TABLE* reversed?

3 2ρι10

[1] [2]
ρtable ←→ 2 3 but
addition sign between rows (to get
colsum, between cols (to get row sums)

+/
sum
reduction
+/ι3
reduces vector to scalar
" matrix to vector
must say
which way
to add -
down or across
+/[1]TABLE
1=rows
adds so
that yet
3 el. in answer
because ~~rows~~
there are 3 rows
in order to do
that, must
add across
the cols -

____

No, the index in brackets refers to the axis along which summation is applied; the shape of the result is the shape determined by the *remaining* axes. Thus, a sum *along* the second axis (columns) yields a sum for each row.

Paradoxically, this matter probably becomes clearer in the case of higher-rank arrays; try some experiments on the rank-3 array *LISTOFTABLES*.

3.5 Is it possible to twist a table so that its rows become columns, or to reverse it so that its columns run backward?

Yes, introduced to ⌽ in lesson 4, refer to manual, experiment

____

Yes, each of the functions ⌽ and ⍉ and ⊖ "flips" its table argument about an axis indicated by the line in its symbol. The symbols are all **composites** formed by overstriking two simple symbols. You may learn how to enter them by experimentation, by consulting the manuals, or by reading instructions in the next session.

3.6 How should one read aloud expressions such as +/[1]*TABLE* and +/[2]*TABLE*, and ⌈/[2]*TABLE*?

____

Various verbalizations are useful. For example, "Column sums" (+/[1]*TABLE*), "Row sums" (+/[2]*TABLE*), and "Row maxima" (⌈/[2]*TABLE*) or "Sums along the first axis", "Sums along the second axis", etc. Expressions using the term "axis" are preferable for higher rank arrays such as tables of tables (rank 4) where suitable terms analogous to "rows" and "columns" may not exist.

4.1 The variable *MONTHS* works because all the abbreviations are the same length, but how could I make a table of the full names of the months?

____

Enter enough spaces after the shorter words to pad them all out to a common length, as in:

```
      X←3 6ρ'JUNE  JULY  AUGUST'
```

4.2 If *MARY*←8 and *X*←'*MARY*', I can see that I cannot expect to write 2×*X* to get 2×8, but it seems to me it should be possible to get the 8 from *X* somehow.

Yes, you can **execute** *X* (using the expression ⍎*X*) as discussed in Session 9. You may also want to consult the manuals.

4.3 Is it possible to make a new table of names by selecting certain of the rows from a table such as *MONTHS*?

Yes, try expressions such as *MONTHS*[1 4 7 10;] (called **indexing**) and 1 0 0 1 0 0 1 0 0 1 0 0/*MONTHS* (called **compression**). These selection functions are treated in Session 10.

4.4 I made a list called *DAYS*, which gives the number of days for each of the twelve months, but I was unable to print them beside the months using either *MONTHS*,*DAYS* or *MONTHS*,' ',*DAYS*.

? error message, look up in manual, the way to do it is ↓

Apply the **format** function ⍕ (whose symbol is called **thorn**) to the one-column table 12 1⍴*DAYS* to produce a 12 by 2 table of literal digits which may be catenated with *MONTHS*, as in *MONTHS*,⍕12 1⍴*DAYS* or (if you want a space between the names and the digits) as in *MONTHS*, ' ',⍕12 1⍴*DAYS*. These matters are treated further is Session 9 and in the manuals.

4.5 In Session 2, I found that I could get *LIST* from *TABLE* by writing 6⍴*TABLE*, but realized that I could write this only if I knew the total number of elements in *TABLE*. I now see that ravel does the same thing, but I still wonder if it is possible to write the corresponding expression using ⍴ for an unknown argument *TABLE*?

----

good questions, generally applicable expressions are frequently more useful

Since ×/⍴*X* gives the total number of elements in an array *X*, you may write (×/⍴*X*)⍴*X* instead of ,*X* for any array *X*.

4.6 What are all of the composite symbols in APL?

Figure 4 of your manual (which you marked in Session 2) shows all of the current composite characters in the right-hand column. However, new ones are occasionally added to the language to denote new primitives.

4.7 I am a manager and came here to learn how to manage APL programmers, **not** to learn to program.

As I said in the introductory lecture, this course is intended primarily as an introduction to APL for people wishing to learn to program in APL. As I also pointed out, a manager can often benefit from some first-hand knowledge of the tools used by his subordinates.

I believe that a good try will convince you that the present course is an efficient vehicle for gaining such first-hand knowledge of APL. However, if you become convinced that this is not the case, you might wish to withdraw.

4.8 I have to advise my management how APL can be used and whether to install it. I do not see how learning to type on an APL keyboard will help me.

____

This course will not provide you with everything you need to make such decisions. However, it should give you enough knowledge of APL to prepare you to communicate effectively with people who do use and manage APL, and so to draw on their experience. Although any typing skill acquired in the course may be of no direct use to you later, the direct epxerience of APL which it makes possible is probably the most effective and lasting way of obtaining the knowledge of APL that you require.

4.9 When are you going to start lecturing. I didn't pay all this money for a few sheets of paper and the use of a fancy typewriter. I realize this class format is a departure from the more usual lecture type of course.

If you review the statement of the objectives of this course distributed at the outset (objectives clearly indicated in publicizing the course) you will see that the course is based heavily on work and experimentation at the terminal, and on associated tutoring. Little use is made of formal lecturing.

The course will be conducted in this spirit, and you will therefore gain little benefit from it if you do not embrace the opportunity to use the powerful computing facilities available at this fancy typewriter.

4.10 Without ever touching the terminal, a student quickly reads through the first session, asks for and gets the second, and then, when the instructor is away from her desk, picks up the remaining sessions and settles down to read them.

____

The instructor should be careful to store the sessions in such a way that students cannot easily obtain any session before completing preceding sessions. However, if this, or any other breach of procedures should occur, the instructor should maintain discipline, and insist that the copies be returned. A student who refuses to obey should, of course, be asked to withdraw.

4.11 I already know six major programming languages and I see no advantage to the one you are now teaching.

____

Because APL differs rather radically from other programming languages, an experienced programmer may take more, rather than less, exposure to APL to be able to appreciate its power and convenience. I can only recommend that you continue through more complex examples of its use before concentrating on comparisons with other languages.

4.12 A student complains that he is grossly mismatched (too fast, too slow, too bright, too stupid) with his partner, and requests a change.

Serious incompatibility between partners at a terminal rarely occurs, but when it does it must be handled with tact. The following options might be considered:

a) Discuss the mismatch with both partners and see what can be done to ameliorate it.

b) If you have noted a similar mismatch (normally in rate of progress) in another pair, you might raise with them the question of a possible switch and, if it is favourably received, discuss the matter with all four.

c) If a spare terminal is available (as it should be), the pair might be assigned to separate terminals. This solution should be adopted only as a last resort because students normally benefit greatly from joint work.

Allowing such a split might move another mismatched but uncomplaining student to now request a change. This can be handled by allowing him to make his own match with one of the two singles now available.

Any techniques for **imposing** good matching of students at the outset (based perhaps on information required from registrants) are unlikely to prove better than the student's own pairing based on common employment or other acquaintance. However, it might be wise to defer the selection of partners until students have made some contact, during coffee served before the specified starting hour, during the introductory lecture, or both.

4.13 My company uses the ABCD application package written in APL, and my job on returning from this course will be to modify and maintain that package. If I finish the sessions quickly, may I then get started on studying the programs in the ABCD application?

I would strongly advise you not to rush through the course. The long-term benefits of the general understanding of APL which you can gain from a thorough use of the sessions and supplementary exercises of the course will far outweigh any short-term advantage of concentrating on the specifics of your first work assignment in APL. Remember also that I will be available to you by telephone for some time after the course to provide help on any APL work you undertake.

At this point, each instructor should take about 20 minutes to prepare a suitable summary lecture covering the material of Sessions 1-4. One should then be asked to present a lecture, and the others should be encouraged to badger him with reasonable questions as students might, so that he learns the dangers of attempting a philosophical review instead of a practical summary.

Slides 12-22 provide a summary of Sessions 1-4. By drawing heavily on the manuals, they strenthen the students familiarity with, and faith in, the manuals, and provide a more lasting summary than would screen images not later available.

Students should be urged to consult and mark the actual pages from which the slides are drawn; browsing in the adjacent material is always helpful.

In using Slides 13-15, point out briefly that the elements used to form **names** and **numbers** are drawn from the alphabets at the top, that the symbols already encountered can be easily found (together with their names), and that the table includes the quote symbol used to form **literals**.

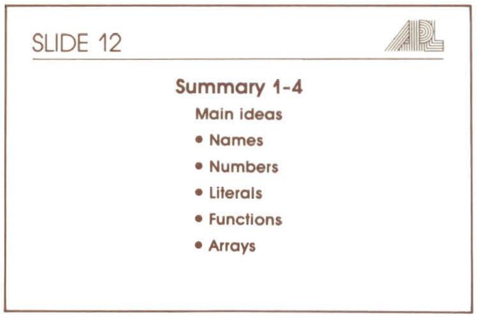
Ask students to identify those symbols already encountered in the course and any others they recognize from other experience, and encourage further browsing in the Character Set table in the manual.

Students should be asked to locate and mark in their manuals the tables referred to in Slide 16.

The notions of functions and arguments are important, and should be discussed (using Slides 17-19) in the light of the following passage taken from page 25 of the IBM manual:

> The word "function" derives from a word which means to execute or to perform. A *function* executes some action on an array (or arrays), called its *argument(s)*, to produce an array as a result. The result may serve as an argument to another function. For example:

```
      3×4
12
      2+(3×4)
14
      (-6)÷3
¯2
```

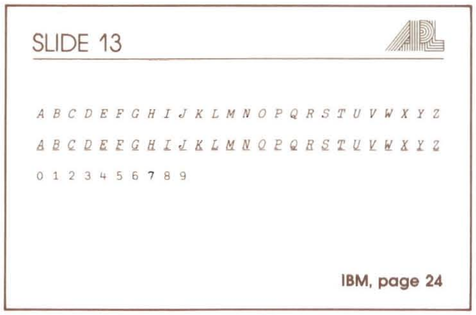


SLIDE 12

Summary 1-4

Main ideas

- **Names**
- **Numbers**
- **Literals**
- **Functions**
- **Arrays**



SLIDE 13

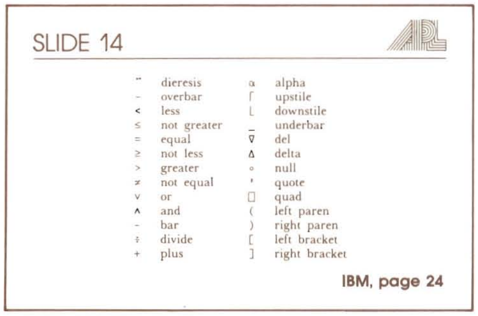
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A̲ B̲ C̲ D̲ E̲ F̲ G̲ H̲ I̲ J̲ K̲ L̲ M̲ N̲ O̲ P̲ Q̲ R̲ S̲ T̲ U̲ V̲ W̲ X̲ Y̲ Z̲

0 1 2 3 4 5 6 7 8 9

IBM, page 24

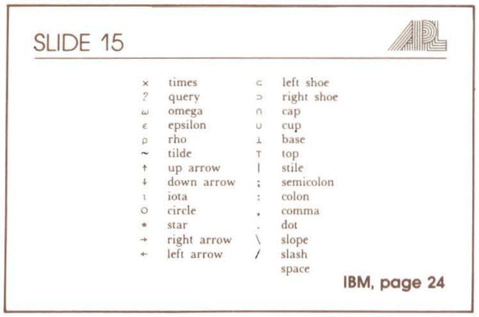
= figure 4



SLIDE 14

¨	dieresis	⍺	alpha
¯	overbar	⌈	upstile
<	less	⌊	downstile
≤	not greater	_	underbar
=	equal	⍫	del
≥	not less	∆	delta
>	greater	∘	null
≠	not equal	'	quote
∨	or	⎕	quad
∧	and	(	left paren
-	bar	)	right paren
÷	divide	[	left bracket
+	plus	]	right bracket

IBM, page 24

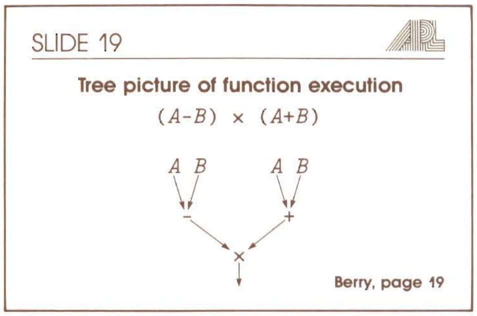


SLIDE 15

×	times	⊂	left shoe
?	query	⊃	right shoe
⍵	omega	∩	cap
∊	epsilon	∪	cup
⍴	rho	⊥	base
~	tilde	⊤	top
↑	up arrow	∣	stile
↓	down arrow	;	semicolon
⍳	iota	:	colon
○	circle	,	comma
⋆	star	.	dot
→	right arrow	\	slope
←	left arrow	/	slash
			space

IBM, page 24

Although the application of scalar functions such as + and × to arrays is readily grasped, the application of other functions such as +/[1] and ,[2] may give students difficulty. Graphic views of the application of such functions can be very helpful, and students should be encouraged to sketch their own views and to look for such views in the manuals. Slides 20-22 illustrate the use of such aids and provide explicit references to material in the manuals.

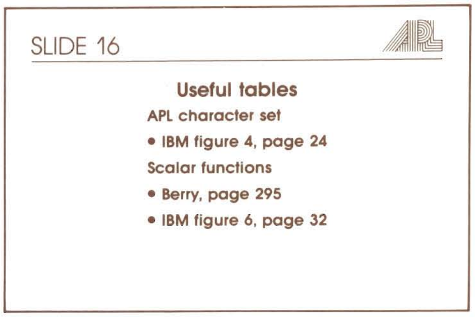


SLIDE 19

Tree picture of function execution

`(A-B) × (A+B)`

Berry, page 19



SLIDE 16

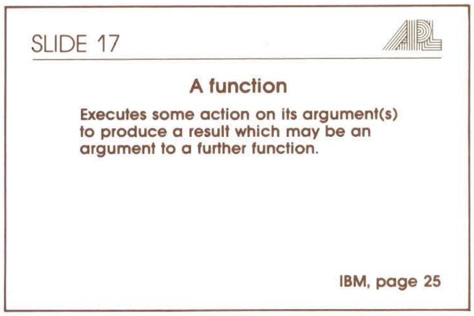
Useful tables

APL character set

- IBM figure 4, page 24

Scalar functions

- Berry, page 295
- IBM figure 6, page 32

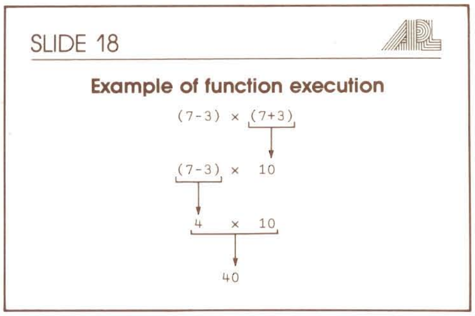


SLIDE 17

A function

Executes some action on its argument(s) to produce a result which may be an argument to a further function.

IBM, page 25

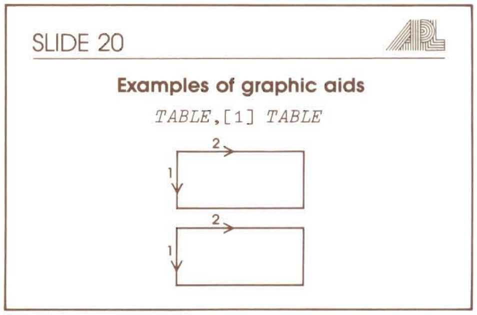


SLIDE 18

Example of function execution

```
(7-3) × (7+3)
(7-3) × 10
4 × 10
40
```

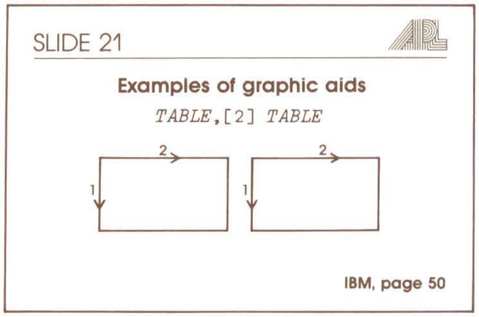
class found these very helpful



SLIDE 20

Examples of graphic aids

TABLE,[1] *TABLE*



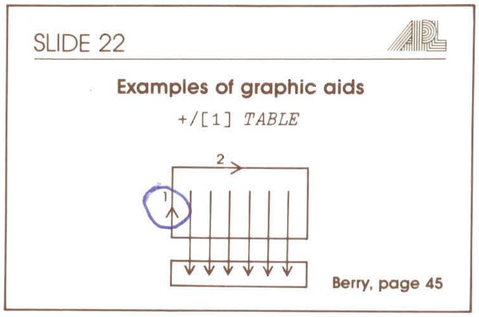
SLIDE 21

Examples of graphic aids

TABLE,[2] *TABLE*

IBM, page 50

✓



SLIDE 22

Examples of graphic aids

+/[1] *TABLE*

Berry, page 45

Instructor's Guide

At this point the instructors should read the Instructor's Guide, and raise any questions concerning its application, with particular reference to their experience in dealing with Sessions 1-4.

LECTURE III
Sessions 5—8

Handling Questions

5.1 Why doesn't)*ERASE LESSON* remove the workspace *LESSON* from my library?

____

The command)*ERASE ABC* erases the name *ABC* from an active workspace; the command)*DROP ABC* drops workspace *ABC* from a library. See the table of System Commands in the manuals.

5.2 Now that I know how to drop a workspace, I suppose I could destroy the workspace for the course by entering)*DROP* 12 *COURSE*.

____

Try entering)*DROP* 12 *COURSE* and note the response. Although you can load a workspace from some library other than your own, you cannot change another's library. Your own library number, which you do not have to supply explicitly in a command like)*LOAD LESSON*, is the first element of the account information ⎕*AI* which can therefore be obtained by entering 1↑⎕*AI*.

5.3 Why do all of these things like)*VARS*, and)*SAVE*, etc., begin with a right parenthesis?

____

Phrases like)*SAVE ABC* are **system commands**, which concern the environment within which you use APL, but which are not themselves expressions in APL. The initial right parenthesis is used to distinguish them from APL expressions, which cannot properly begin with a right parenthesis.

6.1 Can I use underscored names, as in A̲B̲C̲←6?

____

Berry, chapter 7

Yes, A̲B̲C̲ is a valid name. See the manuals for the rules of name formation.

6.2 Is it all right to enter *X*←*SHOW* '*TIMES*', and if so, what is the meaning of *X*?

____

X←*SHOW* '*TIMES*' is a list of characters, a fact that you may test by observing the result of expressions such as ⌽*X*. , ρX

X is not a function (as you may see by trying 3 *X* 4), but can be used to define a function. You can verify this as follows: Erase the function *TIMES* (by entering)*ERASE TIMES*), verify that it no longer exists (by trying 3 *TIMES* 4), then enter *DEFINE X* and again try 3 *TIMES* 4.

6.3 The rule for executing an unparenthesized expression seems senseless.

____

In the IBM manual, the rule is stated in another, but equivalent, way as follows: "the (right-hand) argument of any function is the value of the entire expression to the right". An unparethesized expression can therefore be read either **analytically** from left to right, (the first function encountered at any stage being the overall function to be applied to the result of those to **its** right), or **constructively** from right to left.

These simple rules (which do not admit the complication of **hierarchy** rules — plus before times, etc.) are much easier to use than the more complex rules used in mathematics and in many programming languages. However, their power and convenience will become apparent only after some experience, and it might therefore be better to defer the question for a while.

If, however, you wish to pursue the matter now, you might consult the cited passage in the IBM manual, and read the comparison with mathematical notation beginning on page 461 of "Notation as a Tool of Thought" in the **Source Book in APL**.

7.1 I tried to make a multiplication function called *T* by doing *T←SHOW* *'TIMES'*, but it didn't work.

This is a variant of question 6.2, and the same answer applies.

7.2 I entered *X←SHOW* *'TIMES'* and then *DEFINE* 2↓*X*, but I can't see that it did anything.

If you enter 2↓*X* you will see that the result is a list of characters which define a function called *MES*. Now try 3 *MES* 4 to see that such a function has been established.

Alternatively, display the list of functions (by entering)*FNS*), and look for any new function (in this case *MES*).

7.3 I entered)*ERASE DEFINE* and now I can't seem to define functions.

You can restore the function *DEFINE* in either of two ways:

1. Enter)*LOAD MINE*. This will return your active workspace to the state at which you last saved it under the name *MINE*. If you have defined any new names (functions or variables) since then, you will lose them by this action.

2. Enter)*COPY* 12 *COURSE DEFINE*. This will copy into your active workspace the original function *DEFINE* and will not otherwise affect it.
   mine

7.4 Could I define the function *BEELINE* without using *SQUAREROOT*?

Yes, by using an appropriate expression involving primitive functions only. Try it.

7.5 I was not able to define the function *SUM*.

what have you tried?

---

a) Enter *DEFINE* *'SUM◇+/[α]ω'*.

b) Enter 1 *SUM OIL*72 to test the function *SUM*.

c) Try to gain an understanding of the function definition process by taking the character string which defines *SUM* (that is, +/[α]ω) and substituting for α the left argument used in part b, (that is, 1) and for ω the right argument used (that is, *OIL*72). Enter the resulting expression (that is, +/[1]*OIL*72) and compare the result with the result of 1 *SUM OIL*72.

8.1 I was going to produce a table of abbreviated 4-character country names by entering *EDIT NAMES* followed by slashes under the last five columns, but I got a *DOMAIN ERROR*.

____

You could get the result you expected by various selection functions (to be discussed in Session 10): by indexing (*NAMES* [;1 2 3 4]), by dropping the last five columns (0 ¯5↓*NAMES*), or by taking the first four (8 4↑*NAMES*).

vector

The function *EDIT* is designed to apply only to a list, not to a table. You could apply it to the list obtained by ravelling the table (,*NAMES*) and then reshape the result, as in *ABBNAMES*←8 4 ρ*EDIT*,*NAMES*.

8.2 I tried to get rid of the function *RAP* by revising it and removing the name entirely, but it simply says *NOT DONE* and *RAP* remains in the list of functions.

____

The function *REVISE* simply defines the function represented by the character string given by *EDIT SHOW* ⍞. In your example, you had removed the entire function name preceding the diamond; *DEFINE* was therefore unable to define a function, and so advised you.

A name *ABC* (of either a function or a variable) can be removed by entering)*ERASE ABC* (as shown in Session 5), or by ⎕*EX* '*ABC*'.

8.3 Why does the expression 2 *MAX* 5 7 4 give a domain error? let's look at what 'MAX' is and how it works.

____

Since the definition of *MAX* is *MAX*◇⌈/[α]ω, the expression 2 *MAX* 5 7 4 is equivalent to ⌈/[2]5 7 4. But this expression cannot be executed because 5 7 4, being only a list and not a table, does not possess a second axis (as required by ⌈/[2]) along which to apply the maximum function.

8.4 The function definition used here is not real APL, which I have had some experience of before this class. Why are we using it?

____

The form of function definition you have seen is called the **del** or **canonical** form of definition. It presents a number of complications which obscure the essential simplicity of the important notion of function definition when it is first introduced.

We will also introduce and use the del form of definition, but somewhat later, in Session 16. Interesting examples of its use can be found in the Introductory section of the IBM manual.

8.5 I have used APL a bit before this course and had no difficulty in grasping the canonical form of definition. What are the complications which you say obscure the notion of function definition?

____

Consider the definition *T*:*I*∘×*I*←ιω and the matters that would have to be faced in canonical definition. For example, the header and the purposes of its various parts, the need to localize

I (and therefore the definition of local and global variables), the meaning of the single line number at the left of the display, the meaning of the message *DEFN ERROR*, and finally, the matter of the *SI* stack and "hidden" variables. Direct definition avoids stack problems by producing functions that (like primitives) terminate on any error.

8.6 I have fallen far behind the rest of the class and obviously cannot finish all sessions at this rate. What should I do?

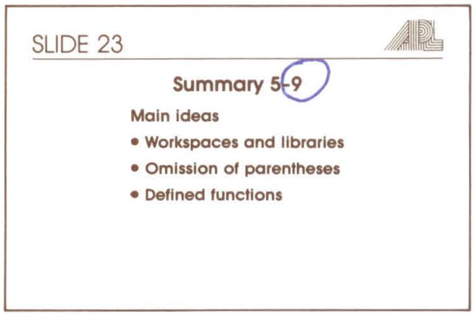
Continue at your own pace. As you have seen, you have been able to work fairly independently so far, and if you continue at the pace at which you are able to absorb the ideas, there is no reason why you should not be able to complete any remaining work when you get home. Furthermore, I will still be available to you by phone when you are completing this work, or even if you embark on other use of APL.

Nothing will be gained by trying to force your pace. At a faster pace you may, in fact, gloss over so many important points that you will find it impossible to proceed at all.

Allow instructors time to prepare and present a lecture as in Sessions 1-4, and then present and discuss the summary on the following slides, guided by the accompanying notes.

By this time, instructors should have picked up and exploited the idea of going to the manuals for material. If not, their efforts should be compared (unfavourably, one would expect) with the slide material which is drawn from the manuals.

Emphasize the fact that an instructor should not be led into prolonged discussion of any of these topics. For details on workspaces, refer the curious student to the manuals. To avoid lengthy arguments on the order of execution, promise to return to the matter at the end of the course when the students will have experienced some of the advantages of the simple APL rule. One may also refer students to treatment of the matter in various articles in the APL Source Book provided for instructors.



SLIDE 23

Summary 5-9

Main ideas

- Workspaces and libraries
- Omission of parentheses
- Defined functions

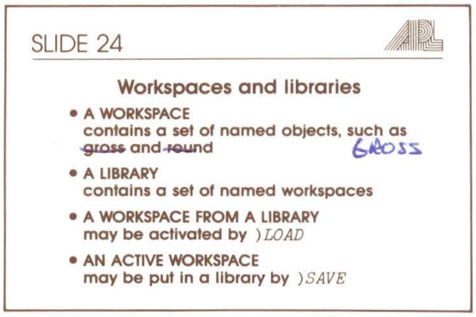
8

Slides 24 and 25 may be used to discuss workspaces and libraries in the light of the following excerpts from the manuals:

> The common organizational unit in an APL system is the workspace. When in use, a workspace is said to be active, and it occupies a block of main storage. Part of each workspace is set aside to serve the internal workings of the system, and the remainder is used, as required, for storing items of information and for containing transient information generated in the course of a computation. (From IBM Manual, page 29)

> An active workspace is always associated with a terminal during a work session, and all transactions with the system are mediated by it. In particular, the names of variables (data items) and defined functions (programs) used in calculations always refer to objects known by those names in the active workspace; information on the progress of program execution is maintained in the state indicator of the active workspace; and control information affecting the form of output is held within the active workspace. (From IBM Manual, page 29)

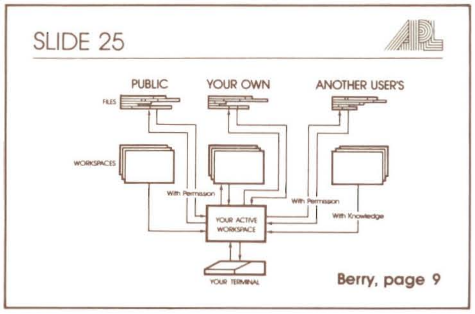
> Inactive workspaces are stored in libraries, where they are identified by arbitrary names. They occupy space in auxiliary storage and cannot be worked with directly. When required, copies of stored workspaces can be made active, or selected information may be copied from them into an active workspace. (From IBM Manual, page 29)



SLIDE 24

Workspaces and libraries

- A WORKSPACE
  contains a set of named objects, such as ~~gross~~ and ~~round~~ GROSS ROUND
- A LIBRARY
  contains a set of named workspaces
- A WORKSPACE FROM A LIBRARY
  may be activated by)LOAD
- AN ACTIVE WORKSPACE
  may be put in a library by)SAVE

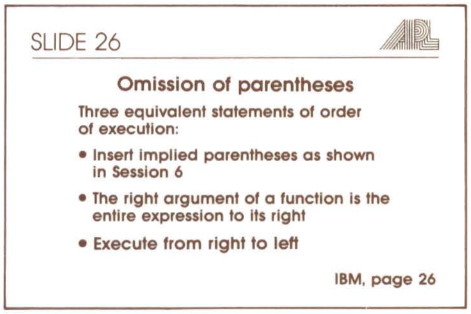


SLIDE 25

Berry, page 9

Slide 26 may be used to discuss the omission of parentheses, in the light of the following excerpts from the manuals:

> Parentheses are used in the familiar way to control the order of execution in a statement. Any expression within matching parentheses is evaluated before applying to the result any function outside the matching pair. (From IBM Manual, page 26)



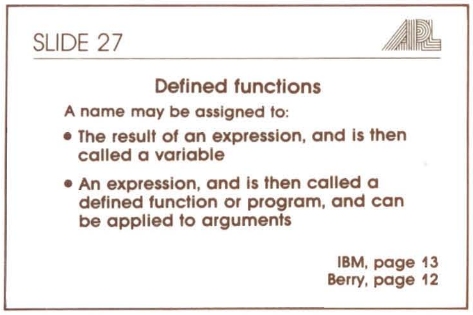
SLIDE 26

Omission of parentheses

Three equivalent statements of order of execution:

- Insert implied parentheses as shown in Session 6
- The right argument of a function is the entire expression to its right
- Execute from right to left

IBM, page 26



SLIDE 27

Defined functions

A name may be assigned to:

- The result of an expression, and is then called a variable
- An expression, and is then called a defined function or program, and can be applied to arguments

IBM, page 13
Berry, page 12

> In conventional notation, the order of execution of an unparenthesized sequence of monadic functions may be stated as follows: the (right-hand) argument of any function is the value of the entire expression to the right. For example, Log Sin Arctan x means the Log of Sin Arctan x, which means Log of Sin of Arctan x. In APL, the same rule applies to dyadic functions as well. Moreover, all functions both primitive and defined, are treated alike; there is no hierarchy among functions (such as multiplication being done before addition or subtraction). (From IBM Manual, page 26)
>
> An equivalent statement of this rule is that an unparenthesized expression is evaluated in order from right to left. For example, the expression `3×8⌈3*|5-7` is equivalent to `3×(8⌈(3*(|(5-7))))`. Their result is `27`. A consequence of the rule is that the only substantive use of parentheses is to form the left argument of a function. For example, `(12÷3)×2` is `8` and `12÷3×2` is `2`. However, redundant pairs of parentheses can be used at will. Thus, `12÷(3×2)` is also `2`. (From IBM Manual, page 26)

Slide 27 may be used to discuss the notion of defined functions, in the light of the following excerpts from the manuals:

> Names are used for two major categories of objects. There are names for collections of data, made up of numbers or characters. Such a named collection is called a variable. Names may also be used for programs made up of sequences of APL statements. Such programs are called defined functions. Once they have been established, names of variables and defined functions can be used in statements by themselves or in combination with the primitive functions and objects. (From IBM Manual, page 13)
>
> Programming is the art of defining new functions that take as their arguments some data you have, and produce as their results some data you want. A program is usually a definition for a function. Presumably, if there already exists a function that does just what you want, you won't need a new definition; so, in practice, programming is writing a definition for a new function that would not otherwise be available to you. (From Berry Manual, page 12)
>
> A definition must be stated in terms that can be understood. The **primitive functions** of a programming language are those functions that need no explanation. (From Berry Manual, page 12)

LECTURE IV
Sessions 9—12

Handling Questions

9.1 Sometimes *A*,[1]*B* works, sometimes *A*,[2]*B* does, and sometimes neither; I don't understand why.

remember, for example 1 2 3 + 4 5 6 7 ←→ length error

____

The shapes of the two arguments to a function must **conform** in certain ways. For example, in the expression *A*+*B*, the shapes ⍴*A* and ⍴*B* must agree unless one of *A* or *B* contains a single element.

The conformability rules for catenation (denoted by the comma) are more complex, but are presented clearly in the manuals. If you wish to experiment, define a tall table *TT*←6 4⍴'⎕', a long table *LT*←4 6⍴'⌹', a long vector *LV*←6 ⍴'○', and a short vector *SV*←4⍴'⊛'. Then display them and try expressions such as *TT*,[1]*TT* and *TT*,[2]*TT* and *TT*,[2]*SV*, etc.

9.2 The experiments with ⍕ and ⍎ suggested in this session involve vectors (such as *N*←123 456 and *C*←'123 456' only. When I try *M*←⍕*OIL*72 or *T*←⍕*OIL*, they work as I expect, but why do ⍎*M* and ⍎*T* give domain errors?

____

haven't decided how it should work.

Although the execute function (⍎) happens to be the inverse of format (⍕) when applied to vectors, its primary purpose is more general, namely, to **execute** its vector argument as an APL expression. For example, ⍎'*OIL*72+*OIL*73' and ⍎'*X*←3'. This definition could be extended to a higher-rank argument in several ways (such as to execute the rows in sequence or, as suggested by your question, to form an array from the execution of all of the rows). The definition is therefore (as yet) restricted to vector arguments.

9.3 How, then, could we obtain the numeric matrix *OIL*72 from the character matrix *M*←⍕*OIL*72?

____

Obtaining the numeric *OIL*72 from the character matrix *M*←⍕*OIL*72 could be done as a sequence of operations, (that is, ⍎*M*[*I*;] for each row *I* of *M*). The definition of functions to perform such sequences will be discussed in Sessions 16 and 18.

Alternatively, you might apply ⍎ to the ravel of *M*, or better (taking a hint from the example ⍎*C*,' ',*C* in Session 9) to the expression ,*M*,' '. It remains to reshape the result *R*←⍎,*M*,' ' by an expression *D*⍴*R*. Note that *D* is neither ⍴*M* nor ⍴*R*, but can be produced by some expression involving both.

10.1 I see how the expressions *U*/[1]*M* and *U*/[2]*M* select rows and columns, respectively, but how could I select both rows and columns.

____

Apply *U*/[1] to the result of *U*/[2]*M*, that is, enter *U*/[1]*U*/[2]*M*. Conversely, you could use *U*/[2]*U*/[1]*M*.

10.2 I have `X←3 5 12 4 7` and understand *U*/*X*, but I inadvertently entered *X*/*X* and got a result I do not understand. (On systems which do not have **replicate**, the question might be that *X*/*X* gave a domain error rather than the "obvious" result actually produced by replicate.)

——————————————————

Try other experiments such as `0 1 2 3 4/X` or `(⍳6)/⍳6`, and try to deduce the general definition of this function, remembering that your definition must cover the **boolean** cases (**zeros** and **ones** only) that occur in the expressions in the session.

You may not find this definition in the manuals, since it is a recent extension of the more limited definition for boolean left arguments. It is not yet incorporated in some APL systems, and may even be incorporated in a system, but not yet in its manual.

10.3 With `X←3 5 12 4 7`, I tried `8↑X`, and similar experiments and I now understand how to extend a vector to the right with zeros. How can I extend it to the left?

——————————————————

Try `¯8↑X`, and similar experiments. You might also try similar experiments on higher-rank numeric arrays (tables, lists of tables, etc.), and also on character vectors and matrices. Alternatively, read about the functions **take** and **drop** in the manuals.

10.4 Is there a simple rule I could use to predict the shape of any array obtained by indexing?

——————————————————

[;]
default
implies
(2) shape

Yes, there is a simple rule for the shape of the result of indexing. Try `⍴M[I;J]` for various shapes and ranks of *I* and *J*, and compare with `⍴I` and `⍴J`. Note that *I* and *J* can themselves be scalars, vectors, matrices, or anything, provided only that their elements are all proper indices to *M*. Confirm your conjecture by consulting the manuals.

10.5 I understand how `(1↓V)-(¯1↓V)` produces differences, and how these differences can be useful. However, I cannot fathom how the function *DIFF* works.

——————————————————

Yes, *DIFF* is difficult, but only because it comprises a **sequence** of things which are individually simple and probably well-understood. One general technique for understanding such a function is to try it for some case (such as `2 DIFF OIL`), then substitute 2 for ⍺ and *OIL* for ⍵ in the definition, and then execute each small part of the expression in the proper sequence.

However, this **reading** technique is the topic of the following session, and you may want to defer further work on *DIFF* to the end of that session.

11.1 I found the `∘.+` very interesting and consulted all three of the tables in the manual that we were told to mark at the outset, but found no clues.

——————————————————

Look for "function tables" in the index in one or both of the manuals.

11.2 What is the point of doubling each quote mark in entering the definition of the function *BARCHART*?

____

Any character used (as the quote is used) to delimit a sequence, cannot itself be included in a sequence without invoking some special convention. A convention often adopted is the one used here for the quote, namely, two quote marks in succession denote an actual quote symbol, not the termination of the sequence.

Consult either or both manuals on this matter. The use of **literal input** (to be discussed in Session 13) avoids the need to double quotes. If you wish to try it in the present context, enter `Z←⍞` followed (in the next entry) by the appropriate character string **without** doubled quotes, that is, the one shown in Session 11 as the result of `Z`. Thus:

```
      Z←⍞
BARCHART◇ '.*'[1+ω∘.≥((ια)÷α)×⌈/ω]
```

11.3 Why is `1` added to *S* in the expression for the barchart?

____

Try the expression with the `1+` deleted. Also review the experiment concerning limitations on indices in Session 10.

11.4 Can you suggest further examples of function definitions to try out the reading techniques of this session?

____

You might try detailed reading of expressions from earlier sessions, although most of them, except possibly the function *DIFF* of Session 10, are probably already too well-understood to be interesting.

Later sessions and the supplementary exercises will provide more challenging examples, and these reading techniques should be applied to them as much as possible.

A wealth of examples can be found in references such as **Notation as a Tool of Thought**, **APL in Exposition**, and **Programming Style in APL**. However, many of these are (like the functions *EDIT*, *DEFINE*, etc., in the workspace `12` *COURSE*) based upon notions introduced in Sessions 16 and 18, and should perhaps not be attempted until these sessions are completed.

12.1 Your definition of "parameter" as "named quantities used in the calculation" conflicts with the normally accepted meaning of the term.

____

As may be seen from the Usage note in the American Heritage dictionary, "the newer nonmathematical senses of **parameter** are widely disputed". The main definition given by American Heritage is "A variable or an arbitrary constant appearing in a mathematical expression, each value of which restricts or determines the specific form of the expression", a definition which accords with the one given in Session 12.

In programming jargon, the term "global variable" is often used in the sense used here for "parameter".

12.2 Your answer concerning the use of the word parameter, seems to denigrate programmers as users of jargon.

Every trade uses jargon, that is, terms having specialized meanings in the trade. In the present instance, "parameter" may be found in any good dictionary, with a definition that accords with that used here, but an entry for "global variable" cannot be found, nor can the intended meaning be properly derived from the individual words "global" and "variable".

12.3 What is the meaning of +\BPI and where is it discussed in the manuals?

The expression +\BPI gives the **cumulative** sums of the elements of BPI. For further information in the manuals, look for the word "scan".

12.4 I don't understand the definition of the function *TAX*.

Review the reading techniques of Session 11 and try to apply them systematically to gain an understanding of the function *TAX*.

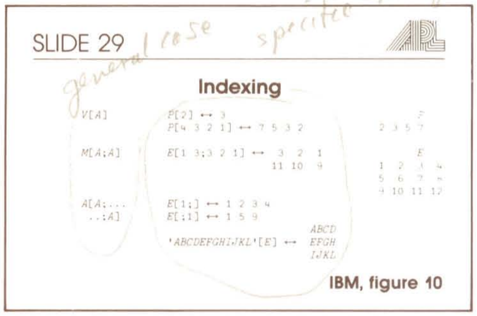
Summary Lecture

A better grasp of the selection functions can be attained with the aid of graphic examples of them. Many are provided in the manuals; Slide 29 shows one of them for the specific case of indexing, and Slide 30 gives points of reference for the other cases.

Slide 31 may be used as a further exercise in reading, first **without** the use of a terminal, and then with the use of the terminal as an aid or as a check. Ask a student to perform the reading.

Function tables are a special case of the use of operators. The instructor should be familiar with the discussion of operators on pages 39-43 of the IBM manual, and should be particularly careful to avoid confusing students by the common practice of using "operator" as a synonym for "function".

Although the notion of operator should be introduced at this point, extended discussion of it should be avoided. An appropriate level of discussion is suggested by Slide 32.



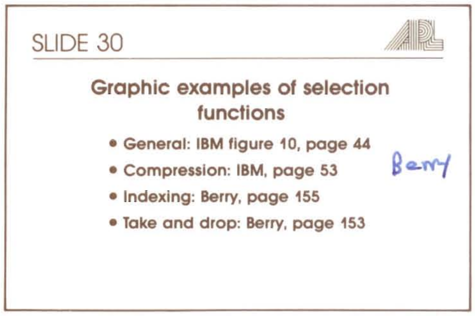
SLIDE 29

general case   specific example

Indexing

```
V[A]        P[2] ↔ 3                                   P
            P[4 3 2 1] ↔ 7 5 3 2                  2 3 5 7

M[A;A]      E[1 3;3 2 1] ↔  3  2  1                     E
                           11 10  9               1  2  3  4
                                                  5  6  7  8
                                                  9 10 11 12
A[A;...     E[1;] ↔ 1 2 3 4
 ..;A]      E[;1] ↔ 1 5 9
                                  ABCD
            'ABCDEFGHIJKL'[E] ↔   EFGH
                                  IJKL
```

IBM, figure 10

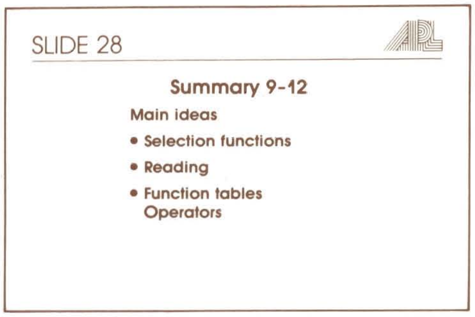


SLIDE 30

Graphic examples of selection functions

- General: IBM figure 10, page 44
- Compression: IBM, page 53
- Indexing: Berry, page 155
- Take and drop: Berry, page 153

Berry 157

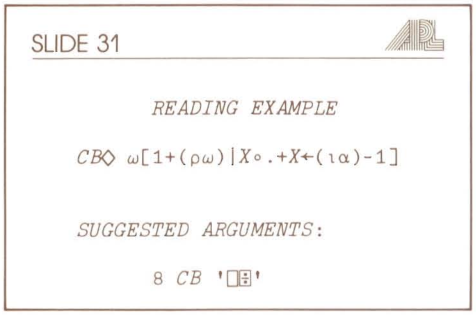


SLIDE 28

Summary 9-12

Main ideas

- Selection functions
- Reading
- Function tables
  Operators



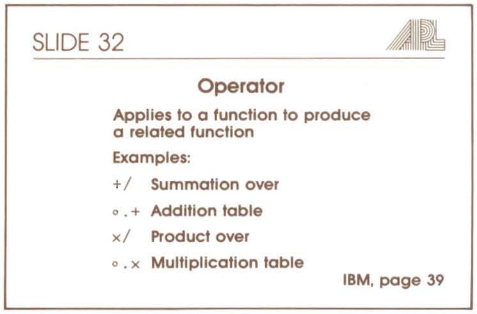
SLIDE 31

READING EXAMPLE

```
CB◊ ω[1+(ρω)|X∘.+X←(ια)-1]
```

SUGGESTED ARGUMENTS:

```
8 CB '⎕⍠'
```



SLIDE 32

Operator

Applies to a function to produce a related function

Examples:

```
+/    Summation over
∘.+   Addition table
×/    Product over
∘.×   Multiplication table
```

IBM, page 39

outer product
∘.f
Berry 189-91

LECTURE V
Sessions 13—16

Handling Questions

13.1 I don't understand the function *KE*.

how it works or what it might be used for.

____

The function *KE* provides another way of accepting keyboard input. Since its definition is dead simple (assuming that you have learned the use of ⍞ and ⍎), your difficulty probably arises from a failure to see the point of the function, that is, its potential use.

It might therefore be best to proceed with the work of this and later sessions in order to see the use of the function, and perhaps to further appreciate its point by trying to substitute other equivalent expressions for it wherever it occurs.

13.2 I don't understand the function *KETAX*.

____

Carefully review the techniques suggested in Session 11, and try to apply them to gain an understanding of the function *KETAX*. If you still feel uncertain about the function, it might be best to make a note to review it again after proceeding through the next three or four sessions.

14.1 Does the use of ⎕← that we have here explain the rather mysterious behaviour of the function *EDIT* that we have been using?

____

Yes, you may want to enter *SHOW* `'EDIT'` to examine the use of ⎕← within it. However, you will still not be able to understand the whole definition of *EDIT* until you understand the **conditional** and **recursive** definition of functions discussed in Session 18.

14.2 What is the purpose of `0↑` in the definition of the function *PR*?

Try this)

____

a) Revise the definition of *PR* so as to remove the execute symbol (⍎), and then experiment with the expression *PR MSG*.

b) Now revise *PR* so as to remove `0↑` as well, and repeat the experiments.

c) Revise *PR* further so as to replace the original `0↑` by `3↑` and repeat the experiment.

14.3 Could the expression `⍎⍞` used in the definition of the function *PR* be replaced by *KE*, using the keyboard entry function defined in Session 13?

____

A good idea. Try it.

15.1 Are there other files available that I could use?

Yes, for example, one of the supplementary exercises (which you can begin after finishing Session 20) concerns the use of a file of geographical data called *NATIONS*. You may experiment with it now if you wish. Since it belongs to account number 13, the name *NATIONS* must be prefaced by `13`, as in `RANGE '13 NATIONS'`.

15.2 I was told that the use of files was more complicated, involving "tieing", "creating", and so on.

Yes, the **primitive** functions provided for handling files are more difficult for beginners to use and we have here provided you with more convenient functions defined using those primitives.

If you ever begin to use files in ways in which more precise control is required (to provide more efficient execution, for example) you will want to learn about the primitive file functions. Study of the use of these primitive functions in the functions provided here will make a good introduction to the topic.

16.1 I got the function *INQUIRY* working all right, but why does it give a value error when I enter `Z←INQUIRY`.

Primitive functions always produce explicit results which can be used as arguments to further functions (as in `2×(3+4)`) or can be assigned a name. A function defined in direct definition form also invariably produces a result, but a function in del form can be defined to produce an explicit result or not, as desired.

Del form definitions which produce explicit results will be found in Session 17. They can also be found in the introductory section of the IBM manual.

The different functional forms possible in del form are determined by the **header**. You may wish to consult the manuals for that term.

16.2 Is there something like the function *EDIT* for making modifications in a function defined in del form?

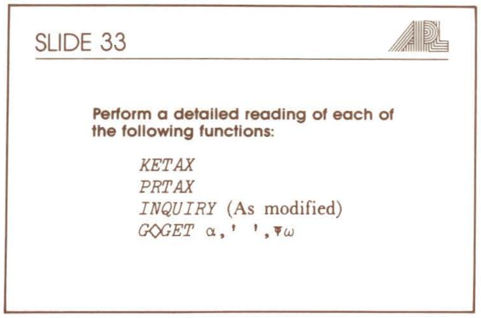
Yes, consult the manuals for "function definition mode". In particular, see the large chart near the beginning of Chapter 10 of the Berry manual, and try to follow it by experimenting on your *INQUIRY* function.

16.3 What kinds of expressions can follow a branch arrow?

Consult the manuals for discussions of "branch", and test your understanding by experiment.

It is often convenient to use **labels** in branch expressions. Consult the manuals for their definition and use. The functions in the introduction to the IBM manual contain examples of their use.

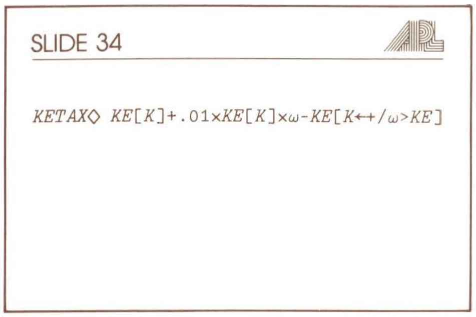
For each of the functions listed on Slide 33 and detailed on Slides 34-37, ask one student to do a detailed reading of it on the blackboard, and allow others to follow by use of their terminals.



SLIDE 33

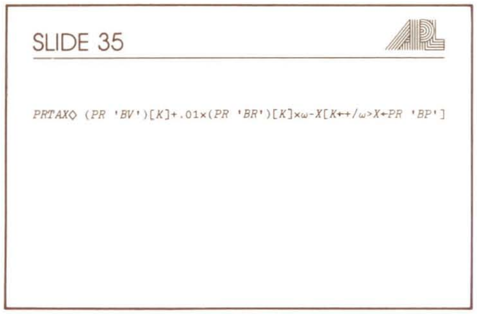
Perform a detailed reading of each of the following functions:

```
KETAX
PRTAX
INQUIRY (As modified)
G◇GET α,' ',⍕ω
```



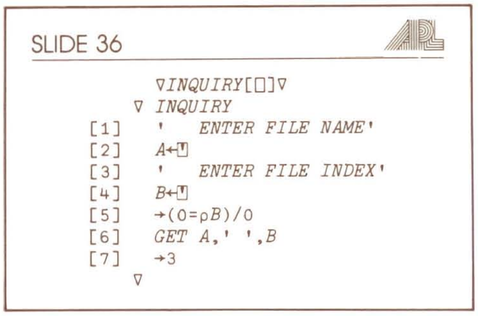
SLIDE 34

```
KETAX◇ KE[K]+.01×KE[K]×ω-KE[K←+/ω>KE]
```



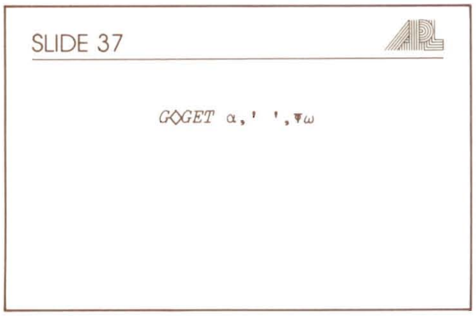
SLIDE 35

```
PRTAX◇ (PR 'BV')[K]+.01×(PR 'BR')[K]×ω-X[K←+/ω>X←PR 'BP']
```



SLIDE 36

```
          ∇INQUIRY[⎕]∇
        ∇ INQUIRY
    [1]   '   ENTER FILE NAME'
    [2]   A←⍞
    [3]   '   ENTER FILE INDEX'
    [4]   B←⍞
    [5]   →(0=⍴B)/0
    [6]   GET A,' ',B
    [7]   →3
        ∇
```



SLIDE 37

```
G◇GET α,' ',⍕ω
```

LECTURE VI
Sessions 17—20

Handling Questions

17.1 What is the *M* produced by *M*←⎕*CR* '*G*'? Is it itself a function?

____

Devise experiments to show that the *M* produced by *M*←⎕*CR* '*G*' is not a function, and that it is simply a table of characters, having the shape 2 7.

You might also enter ⎕*FX M*, 2 2ρ ' *2' to produce a modified definition of the function *G*.

17.2 What is meant by a comment, and how is it used?

____

Look up the word "comment" in the manuals.

17.3 What is ⎕*TRAP*, and why does it occur in direct definitions?

____

⎕*TRAP* is used to "trap" an error, that is, to control what happens in the event that an expression cannot be completely executed. As used here it ensures that a badly-defined function simply terminates with a *DOMAIN ERROR*, and shields the user from many complications associated with "suspension", with the "stack" produced by suspension, etc.

Traps are discussed further in supplementary exercises. You may also wish to consult either manual for the terms "suspension" and "stack". Traps are treated only in the Berry manual.

17.4 Why have two forms of function definition? Can I concentrate on just one of them?

____

Each form of definition has its advantages; every one should at least learn to **read** both, and the serious programmer should learn to write both.

Of their relative advantages, the most important are:

a) Because of its simplicity, direct definition is best for introducing the meaning and use of function definition.

b) Direct definition requires the use of prepared functions (*DEFINE*, *EDIT*, etc., as found in workspace 12 *COURSE*), whereas primitive facilities (⎕*FX* and the "del function editor" discussed in the manuals) are provided for the del form. Primitive facilities for direct definition can be expected to be provided eventually.

c) Because del definition has been in use much longer than direct definition, most current applications are defined in del form, and any modification, or extension, of present applications will need competence in the del form.

d) Because of the clarity and compactness of direct definition, it is commonly used in APL publications, and a reading knowledge of it is needed to make these publications accessible.

e) **Modular** design (i.e., the construction of a complex function as a simple expression which itself employs simpler **component** functions, rather than as a single monolithic whole) usually leads to systems which are easier to understand, to document, and to modify and extend.

> Direct definition encourages modular design: on the one hand, it makes the definiton of simple functions convenient, and, on the other hand, it makes it awkward to string together long sequences of (often tenuously-related) expressions.

Finally, it should be remembered that the uses of the two forms of definition can be intermixed, as they are in workspace `12 COURSE`. In particular, systems defined in del form can be modified or extended using direct definition.

18.1 I find recursive definition difficult.

Yes, many find recursive definition difficult, but it is a powerful design tool well worth learning.

As for anything else, it is probably best to begin by **reading** a lot of recursive definitions prepared by others before attempting to write difficult recursive definitions yourself. In fact, your first **writing** should perhaps be modifications of existing recursive definitions.

Be sure to begin by using the stepwise reading techniques of Session 11, abbreviating or abandoning them only when you are able to grasp larger units directly without more detailed reading.

18.2 But where can I find recursive definitions to read?

If you have already read *FAC*, *FIB*, and *EDIT*, you might best let the matter of recursive definitions rest until you get the Supplementary Exercises, which you will receive when you finish Session 20. These exercises provide further examples of recursive definitions, and references to other sources.

You may also wish to read all of the recursive definitions in the workspace `12 COURSE`.

18.3 I still find it easier to define functions like *FAC* in del form than to use recursion.

Yes, facility in anything as powerful as recursive definition will not be achieved in a day. However, recursive definition may also be used in del form, and you may use your facility in del form to help you to grasp recursive definition.

In particular, take the recursive definitions that you have in direct definition and display their del form definitions, in the manner shown in Session 17. These should show you how to make recursive definitions in del form.

19.1 Is the inner product `+.×` the same as what is called matrix product in mathematics?

Yes `+.×` in APL is the same as matrix product in mathematics, except that it applies to arrays of rank 3 or more as well. Try some examples of `A+.×B` with arrays of rank 3 or more. You will perhaps learn everything needed by observing the shape of `A+.×B` (that is, `⍴A+.×B`) relative to the shapes of *A* and *B*.

19.2 What is the best way to learn about other APL primitives that we have not used in this course?

____

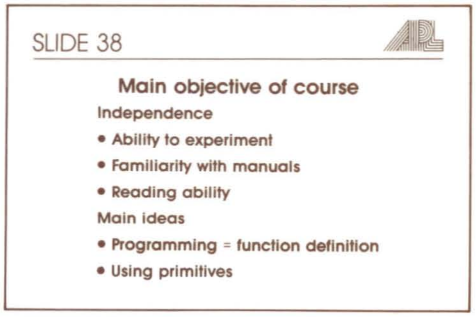
You will find a few new primitives in the supplementary exercises, and you might wish to scan the tables of functions provided in the manuals.

However, it is probably best to learn new primitives by **reading** functions defined by others, using manuals and experimentation when necessary to clarify the definitions of the new primitives encountered. In this way you will see some motivation for each primitive in at least one application of it, and will often learn imaginative programming techniques as well.

20.1 I don't understand the function *SET*.

____

Review the behaviour of the execute function (⍎) in Session 9, and then apply the reading techniques of Session 11 to the expression which defines the function *SET*.



SLIDE 38

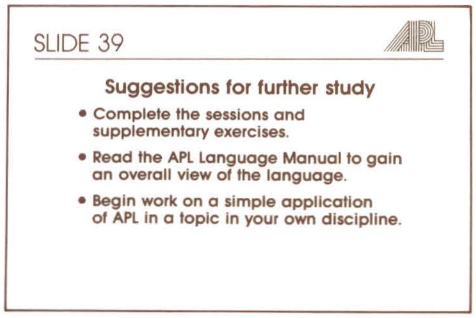
Main objective of course

Independence

- Ability to experiment
- Familiarity with manuals
- Reading ability

Main ideas

- Programming = function definition
- Using primitives



SLIDE 39

Suggestions for further study

- Complete the sessions and supplementary exercises.
- Read the APL Language Manual to gain an overall view of the language.
- Begin work on a simple application of APL in a topic in your own discipline.

Instructors should if possible:

1. Ensure that students are allowed continued access (perhaps with CPU limits applied) to their student account numbers for a week or two following the course, and be sure that they are aware of this privilege.

2. Encourage graduates to call them for help for as long as they wish after completion of the course.

ANNOTATIONS

Supplementary Exercises

The purposes of the supplementary exercises are stated in the instructor's guide as follows:

> Supplementary exercises are provided in a final section. They may be used in various ways: as a source of exercises for review or tests, as material to supplement each session for weaker students or students working independently, or as material for further study following a course. Completion of all of the supplementary exercises would probably require a further three days.
>
> In a brief intensive course the supplementary exercises should be used sparingly, if at all, since the time they take may prevent students from progressing far enough to gain a comprehensive view of the essential topics.

Although an instructor may make little direct use of the supplementary exercises, he should be familiar with them. During the course, they can be used for weaker students to reiterate, in different examples, ideas already presented in the sessions. They can also be used to suggest answers to more probing questions.

Their main use will probably be by students continuing work after the end of the course. Since any help offered by the instructor may then be restricted to telephone or mailbox communication, it is particularly helpful to be acquainted with the exercises.

Doing all the exercises would be too trivial and time-consuming for instructors, and we will instead develop the necessary familiarity by briefly annotating the exercises. A list of suggested annotations is appended, but the instructors should first be asked, in turn, to suggest annotations for the successive exercises.

In the annotations below, the abbreviation R indicates that the exercise simply reiterates ideas presented in the session or earlier exercises, and NF indicates the simple introduction of a new function.

Annotations

1.1 R

1.2 The same function (×) may have widely different interpretations in different areas of application.

1.3 R

1.4 R of 1.2

2.1-2 Combined use of functions (ι with +, -, and ×) to produce obviously useful "grids".

2.3 Input editing. Use if students complain about having to abandon faulty entries. On a system that uses) to recall a previous line, that might also be introduced.

2.4 R of 2.1 (using functions together)

2.5 NF and ambivalent use of symbols

3.1-3 NF and ideas on the utility of higher-rank arrays

4.1-3 R

4.4 NF and answer to question "how can I produce real reports".

5.1 R and new material on libraries

6.1-2 R (Substitution in function definition)

6.3 R (Omission of parentheses)

7.1-3 R (Examples of function definition)

8.1 R and NF (Definition and revision; ○ ω)

9.1-2 R and NF (scan)

10.1-2 R

11.1-3 R

12.1-2 R

13.1 R of Session 11 (Further reading)

13.2 Use of tabs (with forward reference to Session 19)

14.1-3 Further exercise in function definition using keyboard input--introduction of "drill" type functions.

15.1-3 R and the use of established files

15.4 NF (Grade)

15.5-7 Enhanced control of files and introduction to primitive file facilities.

16.1 Further material on ∇ form of definition, including use of labels.

17.1 R

17.2 Introduction of localization of names.

17.3 R

17.4 Further work on ∇ function definition, using a small system so defined.

18.1 Use of reading (Session 11) to fix the notion of recursive definition.

18.2 Sources of further examples of recursive definition.

19.1-2 R

20.1-6 R

20.7 Notion of consulting standard references and translating their statements into APL.

20.8 Further use of reading techniques.

20.9 Introduction to traps via their use to prevent suspension of functions produced by direct definition.

20.10-11 Further work on file primitives.

20.12 Introduction of shared variables by way of their use in the functions for remote tutoring.

They are not his weapons,
but his armor.
W.M. Davis

In his **Geographical Essays***, from which the above quotation is taken, Davis argues that in teaching even the most elementary geography classes, the teacher with a thorough background knowledge of geologic processes can use that knowledge as "a foundation on which to build explanatory stories in his many times of need", and that "... his class will find entertainment and bright instruction, if he addresses them as one who is well-informed beyond the page of his text-book and who knows more than he tells".

But Davis also cautions that "It must be borne in mind that it is not supposed for a moment that every teacher who has learned such facts as these would forthwith teach them to his classes. They are not his weapons, but his armor".

The several papers included in the **APL Source Book** (APL Press, 1981) should be considered as background information to be used as Davis recommends. An instructor thoroughly familiar with them will find them useful in providing immediate answers, and in providing students with further reading; an instructor not familiar with them will run the risk of further contributing to the wealth of misinformation about APL.

The Source Book should be distributed to persons registered for the instructors course so that it may be studied before the course begins. Failing this, it should be distributed on the first day of the course, and its study should be assigned as work outside of class hours.

To ensure familiarity with the material in the Source Book, each instructor should be assigned the task of preparing and presenting a brief report on at least one of the references, in such a manner that all are treated.

Brief model reports appear below.

The Design of APL

> Falkoff and Iverson, IBM Journal of Research and Development, July, 1973. Reprinted in ACM STAPL APL Quote-Quad, Vol. 6, Issue 1, Spring, 1975.

This paper treats the general principles that guided the design of APL, illustrates their application to a number of aspects of the language, and provides (in an Appendix) a brief chronology of the development of APL up to 1973. The authors state the general principles as follows:

> The actual operative principles guiding the design of any complex system must be few and broad. In the present instance we believe these principles to be simplicity and practicality. Simplicity enters in four guises: **uniformity** (rules are few and simple), **generality** (a small number of general functions provide as special cases a host of more

* Davis, W.M., **Geographical Essays**, page 90 in Dover Publications 1954 republication of the 1909 edition.

specialized functions), **familiarity** (familiar symbols and usages are adopted whenever possible), and **brevity** (economy of expression is sought). Practicality is manifested in two respects: concern with actual application of the language, and concern with the practical limitations imposed by existing equipment.

The specific aspects of the language discussed include:

* The **character set** and the keyboard arrangement.

* **Valence** (monadic or dyadic), and the **order of execution** and its consequences.

* **Scalar functions**, with emphasis on the matter of generality (for example, important Boolean functions such as **exclusive-or** fall out as special cases of more generally useful **relational** functions.

* **Operators**, with discussion of the belated recognition of their importance.

* **Formal manipulation** of APL expressions, and the use of identities.

* **Execute**, as a function which "... makes the language 'self-conscious' ...".

* **System Commands** and other environmental facilities.

* **Shared variables** as a generalized communication facility.

The Evolution of APL

Falkoff and Iverson, Session XIV in **History of Programming Languages**, R.L. Wexelblat, Ed., Academic Press, 1981.

This paper discusses the evolution of APL in a manner which complements the earlier "The Design of APL" (IBM Journal of R and D, July, 1975). Although this treatment also includes questions of design choices, it is organized chronologically to give a clearer picture of the lines of development. The periods identified are:

* Academic Use [to 1960], covering the initial motivation as a tool for communication among people, and the double influence of mathematical and programming background.

* **Machine Description** [1961-1963], covering developments stimulated largely by the work of formally describing IBM's System/360 family of machines.

* **Implementation** [1964-1968], covering developments stimulated by the production of a software implementation of the language, including the influence of typewriter restrictions.

* **Systems** [after 1968], covering the environment of an APL program, with emphasis on shared variables as a general communication facility.

The paper concludes with a detailed example (the vector constant) of the evolution of a small facet of the language, and with a discussion of the relation to mathematical notation.

The Inductive Method of Introducing APL

Iverson, **APL Users Meeting Proceedings**, I.P. Sharp Associates, 1980.

This paper treats the philosophy behind the present course, and develops analogies with the teaching of natural languages by the so-called **direct method** used by M.D. Berlitz.

The book by Diller referred to in the paper is excellent, and should be read by any serious teacher of languages.

Programming Style in APL

Iverson, **APL Users Meeting Proceedings**, I.P. Sharp Associates, 1978.

This paper discusses general approaches to improving one's use of APL, and illustrates them by specific examples. It uses direct definition exclusively, and provides some further examples of recursive definition. An appendix provides an explicit translator from direct to del form, itself expressed in del form.

The approaches to improving programming style include:

1. Techniques for assimilating difficult primitives and commonly used phrases. The inner product and the dyadic transpose (which often pose difficulties) are both treated at length in a manner motivated by applications.

2. The advantages of striving for generality in the design of functions, and techniques for achieving generality are discussed at length. The examples used include the application of functions over specified subsets.

3. The uses of identities and proofs are introduced in concrete and fairly simple examples.

4. The importance of reading (in the sense introduced in Session 11) is emphasized, and a number of sources of algorithms for further reading are provided.

Notation as a Tool of Thought

Iverson, **Communications of the ACM**, August 1980.

This paper applies APL to a number of topics to show how it combines the advantages of executability and generality offered by programming languages, with the "thinking" advantages (identities, proofs, etc.) of mathematical notation. It employs direct definition exclusively, provides a number of recursive definitions, and shows their role in inductive proofs.

Many of the examples used will be outside the experience of the non-mathematical student. However, because every example is presented in brief, explicit, direct definitions, any student enjoying the use of an APL terminal should be able to gain a good deal from most of them.

Moreover, many of the topics, such as permutations, directed graphs, and functions on subsets, are of very general utility (even in rather mundane applications in which their utility often goes unrecognized). Consequently, many of the functions presented may be found to be of immediate use, even by people who may not completely understand their internal structure.

Conventions Governing Order of Evaluation

Appendix A of Iverson, **Elementary Functions**, Science Research Associates, 1966

This paper discusses the conventions used in mathematics and compares them with those of APL. The conventions in mathematics are shown to be more chaotic and less well-understood than is commonly realized. It is interesting to note that comprehensive treatments of the established conventions are difficult or impossible to find. For example, Cajori's two-volume **A history of Mathematical Notations** does not address the matter.

Algebra as a Language

Appendix A of Iverson, **Algebra: An Algorithmic Treatment**, APL Press, 1977

Adopting the point of view that "...the algebraic notation taught in high school is a language (and indeed the primary language of mathematics)...", this paper discusses the inconsistencies and deficiencies of algebraic notation, and shows how they are remedied in APL. Analogies between the teaching of algebra and the teaching of natural languages are also discussed.

INSTRUCTOR'S GUIDE

The following statements of objectives and methods of the course are reproduced from the introductory lecture:

> The objectives of this course are:
>
> 1) to give you the ability to translate into APL, and therefore into computer use, procedures of interest in your own profession and known or available to you in terms understood in your profession,
>
> 2) to provide sufficient familiarity with APL reference material and with techniques of experimentation to allow you to continue to expand your mastery of APL through further use and independent study.
>
> The method used allows you to begin work immediately on an APL terminal, experimenting under the guidance of simple written instructions, and proceeding as independently of the instructor as your ability permits. Two students work together at each terminal, to their mutual benefit, and proceed at their own pace independently of other student pairs.
>
> The instructor therefore spends little time in formal lectures, and most of the time in individual tutoring -- answering specific questions (perhaps by directing attention to points covered in earlier sessions), and suggesting general techniques for resolving questions through further experimentation and further use of the reference material.

The approach used differs markedly from most conventional lecture courses; its main characteristics may be summarized as follows:

- It is **practical** in that the student spends most of the time, from the very outset, in practical work at a terminal, guided by simple written instructions. The necessary facility at the keyboard and familiarity with the character set, error messages, etc., are absorbed naturally in the course of addressing more interesting questions.

- It is **inductive** in that the grammar and structure of APL and the characteristics of its primitives are presented largely by examples, rather than by formal definitions.

- It is **heuristic**, not only in the sense that the inductive method leads the student to discover the general rules of the language, but also in presenting and exercising general techniques of investigation, including experimentation at the terminal, critical reading of APL expressions, and experience in the use of manuals and other sources of information.

All of these attributes tend to foster in the student an independence which is more valuable than the more superficial exposure to a wider range of APL primitives that might be achieved in a lecture course. Moreover, they are particularly relevant to APL -- not only does its simple and uniform structure make induction from specific examples easier and the techniques of experimentation more effective, but it makes more practicable subsequent extension of the student's APL vocabulary (i.e., knowledge of APL primitives) through independent use of reference material.

The demands placed on the instructor differ greatly from those imposed by a conventional lecture course. On the one hand, because the few brief lectures are not central to the course, little or no experience or ability in lecturing is required. On the other hand, the instructor needs a greater ability to "lead out" (that is, "educate" in the sense implied by the etymology of that word) students to examine and experiment with the ideas inherent in the material presented, and requires a greater working knowledge of APL.

The greater knowledge of APL is required because the emphasis on exploration may lead students into areas not normally covered in an introductory course, and not necessarily understood by instructors whose knowledge of APL is largely confined to the content of lecture notes.

Because the demands differ from those imposed by a traditional lecture course, instructors are advised to study thoroughly the remainder of this guide, and to adhere closely to the procedures suggested, particularly for the first few trials. Moreover, an instructor should, regardless of the amount of previous experience in using or teaching APL, **go through the sessions on a terminal** as a student would. Finally, it would be wise to review the material in the guide before each of the first few trials of the course.

Conduct of the Course

In order to conduct the course in the spirit outlined in the introduction, the instructor should observe the following specifics:

1) Allow the students to begin work on the terminal with as little delay as possible. In particular:

   a) Have the terminals already signed on to a clear account number (having an empty workspace library and no associated files) and defer any mechanics of dial-up and sign-on until the second day.

   b) Open the session with a brief lecture based on the slides provided; then distribute **the first session only** and allow the students to get to work.

2) Assign two students to each terminal even if extra terminals are available; joint work by students is invariably helpful.

   To allow students to choose suitable partners, it is best to begin with an informal, unseated, period (perhaps with refreshments) in which the need for choosing a partner is announced first, and each student is then asked to introduce himself.

   If a student complains of a mismatch (usually in rate of progress), discuss it with both partners and perhaps suggest a switch with any other mismatched pair you may have noted. If a spare terminal is available (as it should be), a pair can be split. Although pairing is normally advantageous, it is **not essential**. In particular, the course may reasonably be offered to a single student, especially if the instructor has other work to occupy his free time.

3) Some students will read anything in sight rather than start work at the terminal; begin by confining attention strictly to the first session. In particular, do not distribute the reference manuals until they are needed, at the end of the second session.

4) Leave students alone as much as possible to make and correct their own mistakes, and to gain confidence in their ability to find their own way out of difficulties. Answer questions as directly and simply as possible, suppressing the urge to tell more about an issue than the student is prepared to absorb at the time.

5) Allow each student pair to proceed at their own pace. There is no advantage in pushing or constraining them to keep the group synchronized, and no increase in the burden on the tutor due to the lack of synchrony. However, sessions should be handed out one at a time on demand, since this provides added opportunity for contact between student and tutor.

6) Do not rush the sessions, particularly the earlier ones when the students are faced with the most fundamental ideas as well as with an unfamiliar keyboard.

7) Three six-hour days should suffice for the twenty sessions, but it is important not to rush sessions in order to finish them all. Some effort should be made to cover everything up to and including files (Session 15), and perhaps the ∇ form of definition (Session 16), but the remaining sessions should not be considered essential.

   There is, in fact, some advantage in leaving some of the final sessions untouched so that the student has an easy and familiar continuation available to him. Try to assure that the student has immediate access to a terminal when he returns to work; in particular, his access to the student account number should be continued for some period, to encourage him to copy into his own library the workspace he has developed (including the original `12` *COURSE* workspace provided), and to continue his use of APL without interruption.

8) Supplementary exercises are provided in a final section. They may be used in various ways: as a source of exercises for review or tests, as material to supplement each session for weaker students or students working independently, or as material for further study following a course. Completion of all of the supplementary exercises would probably require a further three days.

   In a brief intensive course the supplementary exercises should be used sparingly, if at all, since the time they take may prevent students from progressing far enough to gain a comprehensive view of the essential topics. Nevertheless, the instructor should be thoroughly familiar with the supplementary exercises in order to provide guidance for further study, particularly since some of them (such as 15.5, 15.6, 20.10, and 20.11) lead to more general and more efficient use of some of the functions introduced in the main sessions.

Reference Material

Two reference manuals should be given to each student, one for the core language common to all serious APL systems (of which the best is Reference 1 below), and one for the particular system used (of which the best for SHARP APL is reference 2):

1. **APL Language**, Form # GC26-3847, IBM Corporation.

2. Berry, P.C., **Sharp APL Reference Manual**, I.P. Sharp Associates.

The first reference is also available from IBM in both French (Form # GCF2-0135) and German (Form # GC12-1328).

Each instructor is expected to study the papers in **A Source Book in APL** provided with the course. He should also try to familiarize himself with existing literature on APL, including that cited in the supplementary exercises.

Physical Facilities and Class Size

Because of the tutoring role played by the instructor, the maximum manageable class size depends on the background, interests, and abilities of the students as well as on the experience of the instructor. However, an experienced instructor should be able to handle up to two dozen students. A new instructor should begin as an assistant, or with a class of modest size.

A slide projector and screen should be provided for the prepared slides for the lectures, and a blackboard or overhead projector should be available for handling questions treated during lectures.

A terminal must be provided for each pair of students, and an extra terminal for the instructor is a convenience as well as a desirable back-up in the event of terminal failure.

Any terminal which provides a complete APL character set may be used, but printing terminals are probably to be preferred over video screens for the following reasons:

- They provide printed copy which students may detach for study and revision, or may take away for study after course hours.
- Their usually lower speed is perfectly adequate for the purposes of the course, and discourages playful waste of resources which may occur on entering expressions such as `⍳10000`.
- They commonly provide better resolution so that students are not faced with blurred representations of unfamiliar characters.

However, some printing terminals are noisy and video terminals may therefore be preferred, particularly in crowded classrooms.

Installation of Workspaces

Material for the workspace referred to in the sessions and supplementary exercises as 12 *COURSE* is provided on a tape in the instructor's kit. The instructor must select from this material the workspace suited to the particular APL system in use, and install it under the name 12 *COURSE*. There are three main cases to be considered: use of the SHARP APL time-sharing service, use of an in-house SHARP APL system, and use of a non-SHARP APL system:

1. SHARP APL Time-Sharing system.

The workspace 12 *COURSE* is available, and no further action is required.

2. SHARP APL In-house system.

Install the workspace 12 *COURSE* provided on the tape.

3. Non-SHARP systems.

Install the workspace 12 *NONSHARP* under the name 12 *COURSE*.

Limitations of Particular Terminals and APL systems

Limitations imposed by facilities may force minor changes in the conduct of the student sessions as follows:

1. If the use of library 12 is restricted so that it cannot be used for the course, the instructor must advise students of the library number and name to be substituted for `12` *COURSE*.

2. If either the terminals or the APL system fail to provide the diamond symbol (◇), a substitute symbol must be chosen, and students must be advised of the substitution:

   a) Because some non-SHARP systems do not provide the ◇, the substitution of the colon (:) for the diamond has already been made in the workspace *NONSHARP*.

   b) In the workspace *COURSE*, replacement of the diamond by the colon can be made by a systematic substitution in all of the functions except for the file functions *GET*, *RANGE*, *REMOVE*, and *TO*.

3. Because some APL systems do not provide the necessary facilities for "locking" a function "compiled" from direct definition form in order to make it behave like a primitive, any instructor using the workspace *NONSHARP* should advise students as follows:

   a) Recognize any error message which includes a number within brackets as an indication of a domain error in the function whose name precedes the brackets, and ignore all further detail.

   b) After any such error enter → alone on a line to escape from "suspension of execution". Because the variable ⎕*LC* contains an index for every line in execution, the comparison `0=⍴`⎕*LC* will yield the value `1` if no functions are suspended. It can therefore be used to indicate when further entry of → is needed. Excessive entry of → will do no harm.

4. The file facilities provided by different APL systems differ widely, and often differ from one host operating system to another even for the same APL interpreter. It is therefore impractical to provide the file access functions *GET*, *TO*, etc. in a form that utilizes the file facilities on non-SHARP APL systems.

   The file access functions provided in the workspace *NONSHARP* therefore mimic the behaviour of the corresponding functions in *COURSE* by using variables within the workspace. For the purposes of the sessions (specifically Session 15) the equivalence is complete, and the instructor need make no comment.

   However, the file functions provided in *COURSE* are satisfactory for much general use in file problems (on a SHARP APL system), whereas those provided in *NONSHARP* are not. The student should either be warned of this limitation, or the instructor should modify the functions to utilize the particular file facility available. Although the provision of functions which apply to all such facilities is impractical, adaptation to a particular system should not be too difficult.

5. The default setting of certain parameters such as ⎕*PP* (which controls printing precision) differs on certain APL systems. If the effects of these settings become apparent to students (as they will, for example, in Session 1 when using the IBM 5100 which has a default setting of 5 for ⎕*PP*), the instructor must be prepared to explain the control parameter and either show how to set it to the standard value (10 for ⎕*PP*) or explain how to interpret results which differ from those given in the sessions.

6. The function *DEFINE*, and the functions it employs, have been written with emphasis on clarity rather than efficiency. On micro- or mini-computers, efficiency may become important, and the instructor may wish to replace these functions by equivalent functions designed to execute efficiently on the particular computer in use.

7. Because facilities for correcting errors differs on different systems, the instructor must be prepared to give appropriate advice in Session 4.

8. Slow or otherwise limited workspace file facilities provided by some APL systems may make it inappropriate to use the *SAVE* and *COPY* commands as suggested in Session 5, and the instructor must provide appropriate information. For example,)*SAVE* on an IBM 5120 will be much slower than the somewhat different, but usable,)*CONTINUE*.

Lectures

Lectures should be brief and infrequent, perhaps 15 minutes at the end of each half-day. Moreover, they should be aimed as much at evoking questions as at presenting prepared material. Answers can often be kept brief by suggesting expressions which the students may enter later to further illustrate the behaviour of the functions in question.

The lectures represented by the prepared slides and associated notes are designed to avoid problems posed by the individual pacing allowed the students: for the faster students they serve as reviews and overviews, and for the slower students they serve as general introductions to new material which may help to accelerate their pace. The last lecture discusses avenues of further development.

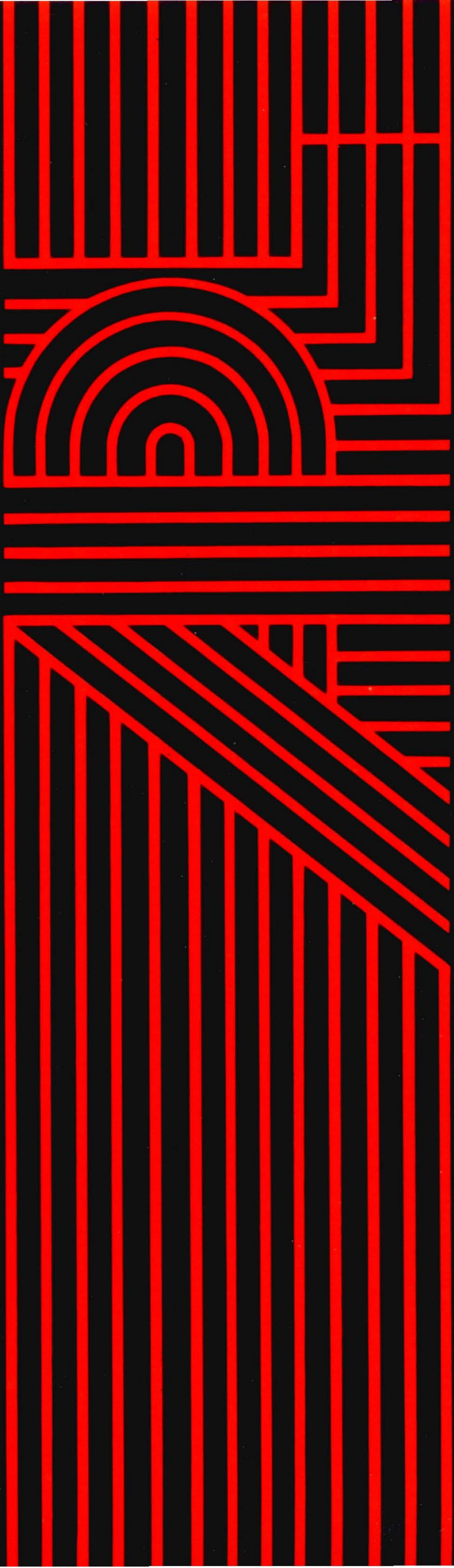
Remote Tutoring Facilities

An instructor may use the function *MONITOR* to converse with a student (at a remote location) using the function *TUTOR* in the manner described by the function *HELP*.

Remote tutoring in the main course should be avoided if possible, but its use in any follow-up period should be encouraged. Instructors should therefore become familiar with the tutoring functions, and should practice their use as well as study their construction.

An instructor using a non-SHARP system may have to adapt the functions, although they employ only the most elementary shared variable facilities. On systems which do not provide shared variables, communication could be provided by way of files.

The coordination provided by the message facilities can, of course, be provided by alternate means, including even ordinary telephone calls.



STUDENT SESSIONS

Table of Contents

1a

The left side of each page provides examples to be entered on the keyboard, and the right side provides comments on them. Each expression entered must be followed by striking the RETURN key to signal the APL system to execute the expression.

```
      AREA←8×2                       The name AREA is assigned to the result
      HEIGHT←3                       of the multiplication, that is, 16
      VOLUME←HEIGHT×AREA
      HEIGHT×AREA                    If no name is assigned to the result, it
48                                   is printed
      VOLUME
48
      3×8×2
48
      LENGTH←8 7 6 5                 Names may be assigned to lists
      WIDTH←2 3 4 5
      LENGTH×WIDTH
16 21 24 25
      PERIMETER←2×(LENGTH+WIDTH)     Parentheses specify the order in which
      PERIMETER                      parts of an expression are to be
20 20 20 20                          executed

      1.12×1.12×1.12                 Decimal numbers may be used
1.404928

      1.12*3                         Yield of 12 percent for 3 years
1.404928

      C←2000×(1.12*3)                Capital from investment of 2000
      C
2809.856
      ⌊C                             Rounded down to nearest dollar
2809
      ⌊.5+C                          Rounded to nearest dollar
2810
      (⌊C×100)÷100                   Rounded down to nearest cent
2809.85
```

NOTES

1a

Names and Expressions

1b

The purpose of a specific example is to provide a basis for grasping the idea of the general case. Thus the example `1.12*3` should suggest that any pair of numbers may be used with the **power** function denoted by the asterisk. What, then, are the meanings of `9*2` and `9*.5` and `6*.5`? What is the meaning of `⌈`, a symbol not yet used, but showing a graphic similarity to the symbol `⌊`?

Review the foregoing examples with a view to grasping the general cases they exemplify, trying your own examples to test your conclusions.

Terminology. Symbols such as + and × denote **functions** which apply to **arguments** to produce **results**. Thus the expression `8÷2` means that the **divide** function, denoted by the symbol ÷, is to be applied to the left argument `8` and the right argument `2` to produce the result `4`, which is the quotient of `8` divided by `2`. The essence of using a computer lies in learning the meaning of a relatively small number of given **primitive** functions such as + and ×, and learning how to use them in combination to construct and define new functions suited to perform the tasks required in a particular profession.

NOTES

Berry 57, 58, 59
symbols

2a

It is often easiest to gain an understanding of a function or expression by using it in a few examples, and then attempting to state briefly in English what it does. Beside each expression below, describe the point that it illustrates. For example:

A) The function denoted by ι (Iota) produces a list of integers up to its argument.

B) *S*×ι*N* produces a list of *N* elements spaced *S* apart.

```
      ι10

      .2×ι10

      3+.2×ι10

      LIST←ι6
      TABLE←2 3 ρ LIST
      LIST×LIST

      TABLE×TABLE

      2×TABLE

      LISTOFTABLES←2 3 4ρι24

      5ρ1 0
      5 5ρ1 0
      4 4ρ1 0 0 0 0

      ρLIST
      ρTABLE
      ρLISTOFTABLES

      6-ι11
5 4 3 2 1 0 ¯1 ¯2 ¯3 ¯4 ¯5

      ¯2×(6-ι11)
¯10 ¯8 ¯6 ¯4 ¯2 0 2 4 6 8 10
```

NOTES

Terminology. The symbols + and ×, which are called St. George's Cross , and St. Andrew's Cross, respectively, denote the functions **plus** and **times**. In reading expressions using them it is best to use the name of the function denoted, rather than the name of the symbol, as in "three plus four", not "three St.George's Cross four".

Similarly, the expression `2 3 ρ LIST` should normally be read as "2 3 reshape (of) *LIST*" rather than "2 3 rho *LIST*" (ρ is the Greek letter rho), and `LIST,7 8` should be read as "*LIST* catenated with 7 8", or simply as "*LIST* with 7 8". Learning the function names as well as (or instead of) the symbol names facilitates both reference and understanding.

* Put markers in your APL Language IBM reference manual at the following important tables, and mark corresponding information in your second manual:

p.24 Figure 4: The APL Character set
p.32 Figure 6: Primitive Scalar Functions
p.44 Figure 10: Primitive Mixed Functions

* Use the manuals to give the names of each of the new symbols encountered in this session, and the names of the functions they represent.

NOTES

3a

Beside each expression below, write a brief statement of what it does. If necessary, consult the hints which follow certain groups of expressions:

```
+/LIST
```

Berry p.185
reduction operator

```
⌈/LIST

⌈/3 1 4 2

⌊/LIST

+/[1]TABLE

+/[2]TABLE

A←LISTOFTABLES

+/[1]A

+/[2]A

+/[3]A
```

Hint: Examine the shapes of the arguments and results, that is, ρA and ρ+/[1]A, etc.

```
LIST,LIST
```

A, [x] List ↔ error

```
TABLE,[1]TABLE
```

A, [3] table ↔

```
1  2  3  4  1
5  6  7  8  2
9 10 11 12  3
     etc
```

added as
another column

```
A,[1]A

A,[2]A

A,[3]A
```

Hint: Examine the shapes of the arguments and results.

NOTES

3a

Further Experimentation

3b

* Give the names of the new symbols encountered in this session, and the names of the functions they represent.

* Enter ?6 several times and try to determine the use of the question mark. Then enter expressions of the form 2 3⍴9 and *M*←?2 3⍴9 to examine its application to a table.

Hint: ? is a random-number generator. It is handy for producing arbitrary lists and tables for use in further experiments. For example, display *M* and repeat some of the earlier experiments on it.

NOTES

3b

Further Experimentation

```
      JANET←5                           Janet received 5 letters today
      MARY←8
      MARY⌈JANET                        The maximum received by one of them
8
      MARY⌊JANET                        The minimum
5
      MARY>JANET                        Mary received more than Janet
1
      MARY=JANET                        They did not receive an equal number
0
```

What sense can you make of the following sentences:

JANET has 5 letters and *MARY* has 8

JANET has 5 letters and *MARY* has 4

'*JANET*' has 5 letters and '*MARY*' has 4

The last sentence above uses quotation marks in the usual way to make a **literal reference** to the (letters in the) name itself as opposed to what it denotes. The second sentence points up the potential ambiguity which is resolved by quote marks.

```
      LIST←24.6 3 17
      ⍴LIST
3
      WORD←'LIST'
      ⍴WORD
4
      SENTENCE←'LIST THE NET GAINS'
      ⍴SENTENCE
18
      GIBBERISH←'JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC'
      MONTHS←12 3⍴GIBBERISH
      MONTHS
```

NOTES

4a

Literals

```
⌽SENTENCE
```

The **composite** symbol ⌽ is formed by ○ BACKSPACE |
See the manuals

```
⌽[1]MONTHS
⌽[2]MONTHS
```

with a scalar and an array [X] applies to the array

with 2 arrays, [2] applies to the array w/ greatest rank

```
'*',[2] MONTHS
' ',[2] MONTHS
```

eg. ⍴X ⇔ 2 4 , ⍴Y ⇔ 2 3 4

X,[2]Y , adds X as a row, [2] to Y, +

Y,[2]X, adds X as a row, to Y.

```
,MONTHS


,' ',[2] MONTHS
```

* The comma is used here to denote both a **monadic** function (which applies to one argument), and a **dyadic** function (which applies to two arguments). Give the names of the functions denoted by comma for the two cases.

* Ravel *TABLE* and *LISTOFTABLES*.

Facilities for the correction of errors differ slightly on different APL systems. If yours does not behave as described below, you may wish to ask the instructor for help.

Because omission of the closing quote is a common error, you are given an opportunity to add it (after an intervening carriage return) as illustrated below:

```
      S←'NOW IS THE TIME
EDIT OPEN QUOTE
      S←'NOW IS THE TIME
                                             Enter carriage return alone here
      S←'NOW IS THE TIME'                    Enter closing quote and carriage return
```

Striking the **linefeed** key erases anything at and to the right of the position of the carriage (or cursor). Corrections can therefore be made by backspacing to any point, striking linefeed, and continuing typing. Experiment with this facility.

* Review Sessions 1-4 and summarize for your own use the material covered.

Summary Lecture | Slides 12-22

NOTES

SYNTAX ERROR
Release 17

4b

Literals

5a

The **workspace** in which your work is carried out by ~~the computer~~ APL contains a record of the names used and of the quantities assigned to them. It also keeps a catalog of these names (or **variables**, as they are often called) which can be obtained by entering:

```
      )VARS
```

This command is useful to remind you of names used or to help you explore a workspace prepared by someone else

```
      )SAVE LESSON
```

A copy of this active workspace will be saved in your **library** under the name *LESSON*

```
      )LIB
LESSON
```

Listing the names of workspaces in your library verifies that it contains the workspace *LESSON*.

```
      PERIMETER
20 20 20 20
      )VARS
```

Verify that the active workspace is not destroyed by saving a copy of it

```
      )CLEAR
      )VARS
```

Clear the active workspace, and verify that it has been cleared

```
      )LOAD LESSON
      )VARS
```

Load the library copy and verify that the active workspace has been restored

```
      )ERASE VOLUME
```

Erase the name *VOLUME*

```
      VOLUME
VALUE ERROR
      VOLUME
      ^

      )COPY LESSON VOLUME
SAVED  18.10.21 01/31/80
      VOLUME
48
```

NOTES

5a

A Library of Workspaces

```
      )ERASE VOLUME

      )LOAD LESSON
SAVED 18.10.21 01/31/80
      VOLUME
48

      )LIB 12
COURSE
```

List the workspaces in public library 12

```
      )LOAD 12 COURSE
SAVED  18.22.24 03/24/80
```

Load *COURSE* from library 12

```
      )SAVE MINE
```

Add the workspace 12 *COURSE* to your library under the name *MINE*

```
      )LIB
LESSON  MINE
```

List your library of workspaces

6a

```
      )LIB
LESSON  MINE
      )LOAD MINE
      )VARS
CAPITAL DEFTEST GROSS   ITEMS   NAMES   OIL     OIL72   OIL73
RATE    SALES80 YEARS   ALPH    FORMS   TRAP
```

List your library of workspaces, load *MINE*, and explore its variables

This workspace also contains **user-defined** functions which supplement the **primitive** functions such as +,×,⋆, and ⌊:

```
      )FNS
AT      DEFINE  DELETE  DESCRIBE        EDIT    FLIB    GET
HELP    MONITOR PRALL   PRINT   RANGE   RC      REMOVE  ROUND
SHOW    TIMES   TO      TUTOR   CFD     EQ      FORM    LCL
LOCK    SPLIT   ~~SQZ~~  NOT PRESENT
```

```
      GROSS
24.783 31.146 42 29.56
      ROUND GROSS
25 31 42 30
```

The function *ROUND* does what its name suggests

```
      RC GROSS
24.78 31.15 42 29.56
```

RC Rounds to nearest cent

```
      YEARS AT RATE
1.404928
```

The function *AT* determines yield

```
      CAPITAL TIMES YEARS AT RATE
2809.856

      SHOW 'ROUND'
ROUND◇ ⌊.5+ω
```

Show the definition of the function *ROUND*

```
      SHOW 'RC'
RC◇ (ROUND ω×100)÷100
```

NOTES

α + ω ≡ placeholders for variables

```
      SHOW 'AT'
AT◇ (1+ω÷100)*α
      SHOW 'TIMES'
TIMES◇ α×ω
```

Read this definition of *AT* as "Divide the right argument by 100, add 1, and raise the result to the power given by the left argument"

* Review the comments on terminology in Session 2, and state how the symbols α and ω (the first and last letters of the Greek alphabet) should be read when they occur in function definitions.

* Enter *HELP* to obtain instructions for communicating with a remote tutor, and use the manuals to clarify any new terms encountered. If you have not been assigned a remote tutor you may wish to skip this.

OMIT

Omission of parentheses. If an expression is not completely parenthesized, then it is to be executed in an order as if parentheses are inserted beginning at the right , as shown below. Until this rule becomes familiar, one should mentally, or actually, insert the full set of parentheses.

* Enter *A*←10 and *B*←13. Then enter some of the following expressions to verify that they are equivalent:

```
A+3×ιB-6

     ↓   ↓
A+3×ι(B-6)

    ↓      ↓
A+3×(ι(B-6))

  ↓           ↓
A+(3×(ι(B-6)))
```

← for indicators only, not to be typed

NOTES

```
      SHOW 'ROUND'
ROUND◇ ⌊.5+ω
      DEFINE 'R◇ ⌊.5+ω'                   Define an equivalent function
                                          having a shorter name
      R 12 13 14÷3
4 4 5

      DEFINE 'SQUAREROOT◇ ω⋆.5'

      DEFINE 'BEELINE◇ SQUAREROOT (α⋆2)+(ω⋆2)'

      3 BEELINE 4
5
      DEFINE 'HYPOTENUSE◇ αBEELINEω'
```

Except in the simplest cases, function definition should be approached in the following steps:

1) Choose some sample values for which the desired result of applying the function is known.

2) Enter an expression expected to give this result, and revise it until it does.

3) Define the desired function, using the final correct expression, but with α substituted for the left argument, and ω substituted for the right.

For example, define a function called *ROWSUMS* which applied to a table yields the sums of the rows. As an example, use the table *OIL*72 which gives oil imports from each of eight countries for each month of the year 1972. We first try:

```
      +/[1]OIL72
```

and see that it gives a twelve-element result. Since *OIL*72 has 8 rows and 12 columns (as given by ⍴*OIL*72), this result cannot be correct, and we revise it to:

```
      +/[2]OIL72
```

We now define *ROWSUMS* by this expression, with ω substituted for the argument name *OIL72*. Thus:

```
      DEFINE 'ROWSUMS◇ +/[2]ω'
```

* Try the function *ROWSUMS* on various tables.

* Define and use a corresponding function *COLSUMS*.

* Define a function *SUM* such that 1 *SUM T* gives column sums of a table *T*, and 2 *SUM T* gives row sums of *T*.

Hint: Try the following sequence:

```
A←1
+/[A] OIL72
A←2
+/[A] OIL72
```

* Try the function *SUM* on the argument *OIL* with various left arguments 1, and 2, and 3. State the general behavior of the function.

* The cost of a list of order quantities *Q* whose prices are *P* is obtained by multiplying each quantity by the corresponding price and summing the results. Define a function *CAT* (Cost At) to give this result. For example, if

```
P←20 15 10 24
Q←5 4 0 2
```

then *Q CAT P* should give 208.

NOTES

* Enter

```
        )SAVE
```

at the end of each session to save the current state of your workspace, and hence to save any new functions you may have defined. The command)*SAVE* saves a copy of the active workspace in the library under its current name. At this point this name would normally be *MINE*, but this name could have been changed to *CONTINUE* by a line drop, that is, a break in communication with the central computer. For further information, locate the reference to "line drop"in the index to the manual used.

NOTES

8a

It is often necessary to revise the definition of a function several times before getting it right. To facilitate this we will now introduce a function *EDIT* which allows convenient insertions and deletions in the argument of characters to which it is applied: slashes delete the characters under which they appear, and anything following the first comma is inserted ahead of the position occupied by the comma. For example:

```
      TEXT←'DDELLLETN AND INSRTION'
      Z←EDIT TEXT                   Apply EDIT to erroneous text
DDELLLETN AND INSRTION                    Line printed by the function
/   //  ,IO                         Line entered on keyboard
DELETION AND INSRTION                     Line printed by the function
                ,E                  Line entered on keyboard
DELETION AND INSERTION                    Line printed by the function
                                    Empty line entered on keyboard (carriage
                                    return alone) ends execution of EDIT


      Z                             The result of the function is the final
DELETION AND INSERTION              corrected line printed
```

Revision of a function.

```
      SHOW 'RC'
RC◇ (ROUND ω×100)÷100
      Q←EDIT SHOW 'RC'
RC◇ (ROUND ω×100)÷100
//,RAP
RAP◇ (ROUND ω×100)÷100
                /,⋆α
RAP◇ (ROUND ω×10⋆α)÷100
                      /,⋆α
RAP◇ (ROUND ω×10⋆α)÷10⋆α

      Q                             Q is simply a character string; the
RAP◇ (ROUND ω×10⋆α)÷10⋆α            function RAP becomes defined only when
      DEFINE Q                      the DEFINE function is applied to it
      3 RAP 3.45678
3.457
```

NOTES

```
      DEFINE 'REVISE◇ DEFINE EDIT SHOW ω'          Define a function for revision

      REVISE 'SUM'
SUM◇ +/[α]ω
///,MAX
MAX◇ +/[α]ω
     /,⌈
MAX◇ ⌈/[α]ω

      2 MAX OIL72
6880 10359 30193 6509 5674 5310 5219 21079
```

*   Apply the function *MAX* to various arguments

*   Define an analogous function *MIN*.

The definition of the function *EDIT* may be displayed in the usual way by *SHOW 'EDIT'*. However, because it uses some primitives not yet introduced, its discussion will be deferred to a later session. Save your workspace at the end of this session.

*   Extend your summary of notation to cover material to the end of this session.

Give Summary Lecture
5-8

Slides 23-27

NOTES

8b

Revising Functions

9a

The following pairs of experiments illustrate the fact that literal digits behave differently from the numbers that they represent; Enter the expressions which appear on the same line as two seperate entries, one after the other

```
N←123 456                    C←'123 456'
ρN                           ρC

2×N                          2×C

N,N                          C,C

N,C                          C,N
```

The following experiments illustrate the use of a function which applies to numbers and produces the character strings which represent them, and an inverse function which "executes" character strings to produce the numbers represented:

```
CA←⍕N                        NA←⍎C
ρCA                          ρNA

2×CA                         2×NA

CA,CA                        ⍎C,C

                             ⍎C,' ',C
```

Berry p.170 execute

Berry 173-178 format

NOTES

9a

Literal Digits

9b

Reports may be produced by catenating literal textual information with the literal table representing the numeric data:

`SALES80`	Display table of monthly sales for 1980
`)COPY LESSON MONTHS`	*MONTHS* was specified in Session 4 and saved in Session 5
`MONTHS,[2] ⍕SALES80`	
`B←MONTHS,[2] '|',[2] ⍕SALES80`	[x] index applies to the array of greatest rank eg. ⍴x ↔ 2 4, + ⍴y ↔ 2 3 4, then x,[2] y adds x as a row to top of y, + y,[2] x adds x as a row to bottom of y
`B`	
`ITEMS`	
`ITEMS,[1] ' ',[1] B`	
`NAMES`	Names of oil exporting nations
`NAMES,[2] ⍕ OIL72`	A report of oil imports

* Define a function called *REPORT* such that the expression

```
NAMES REPORT OIL72
```

yields a report for 1972.

NOTES

9b

Literal Digits

```
      X←3 5 12 4 7
      U←1 0 1 0 1
      U/X
3 12 7
      ~U
0 1 0 1 0
      (~U)/X
5 4
      (X>4)/X
5 12 7
```

Berry p.157
compression

```
      M←5 5ρι25
      U/[2]M

      U/[1]M

      Z←OIL73

      C←(+/[2] Z)>150000

      C/[1] Z

      C/[1] NAMES

      2↑X
3 5
      2↓X
12 4 7
      2 3↑M
1 2 3
6 7 8
```

Berry, p.153
take, drop

* Determine the names of the selection functions introduced above.

NOTES

10a

Selection Functions

Berry p.155 – 157

```
      X[3]
12
      X[2 4]
5 4
      M[2 4;1 3 5]
 6  8 10
16 18 20
      M[2;]
6 7 8 9 10
      M[;1 3]
 1  3
 6  8
11 13
16 18
21 23
      M[;3]
3 8 13 18 23

      ⍴ALT←OIL[1 2 3 4;;]
4 8 12
```

Years 72 to 75 (Compare *ALT* with *OIL* and with *OIL*72)

```
      ALT[;2 3;9+⍳3]
10359  8613  8249
27380 25899 30193

15976 13187 16371
29789 28693 25908

21819 30373 20211
22652 21536 25192

23368 23639 24077
20881 18904 19733
```

* Do experiments such as *X*[0] and *X*[8] to determine the limits on meaningful indices.

* Try expressions such as '*ABCDEF*'[3 1 6 5] and '.*'[2 1 2 2 1 2]

NOTES

10b

Selection Functions

←

```
ALPH←'ABCDEFGHIJKLMNOPQRSTUVWXYZ '

ALPH[X]

ALPH[M]

V←2 3 5 7 11 13 17

1↓V

¯1↓V

(1↓V)-(¯1↓V)                  Differences between successive elements

B←OIL72

0 1↓B                         Drop first column of B

0 ¯1↓B                        Drop last column

(0 1↓B)-(0 ¯1↓B)              Difference gives month-to-month change

D←1 0 0

(D↓OIL)-((-D)↓OIL)            Year-to-year changes

D←0 0 1

(D↓OIL)-((-D)↓OIL)            Month-to-month changes
```

NOTES

```
      A←1

      ρOIL
7 8 12
      ρρOIL
3
      ιρρOIL
1 2 3
      D←A=ιρρOIL
      D
1 0 0

      DEFINE'DIFF◇ (D↓ω)-(-D←α=ιρρω)↓ω'

      2 DIFF OIL
```

* Use the function *DIFF* with the right argument *OIL* and with various left arguments, and interpret the results.

* Apply the function *DIFF* to various vector (list) right arguments, and interpret the results.

```
      )SAVE
```

NOTES

To read a simple but unfamiliar expression such as A∘.+B (that is, to grasp its meaning), apply it to various arguments, and try to abstract the meaning from the results:

Berry 189-91 outer product

```
      A←2 3 5
      B←4 3 2 1

      A∘.+B
6 5 4 3
7 6 5 4
9 8 7 6
```

Also try related expressions such as A∘.×B and A∘.<B

* The tables produced by the foregoing outer products are **function tables** in the sense of **addition tables** and **multiplication tables** used in elementary school. Border the tables by their arguments so as to make clear the calculation that produces them. Make the table by entering a suitable expression, and then border it by hand, as illustrated below for the expression C∘.+C←ι4~~.~~

←

+	1	2	3	4
1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8

NOTES

To read a more extensive expression, begin in the same way by applying it to suitable arguments to get a notion of its overall behavior, and continue by applying parts of the expression in the order in which they are executed in the overall expression:

```
      N←25
      A←NAMES
      B←+/[1]+/[3]OIL
      B
2009404 1886577 1512669 935940 898471 817481 808492 2500696

      A,'.*'[1+B∘.≥(⌈/B)×(⍳N)÷N]
ARABIA   ********************.....
NIGERIA  ******************.......
CANADA   ***************..........
INDONESIA*********................
IRAN     ********.................
LIBYA    ********.................
ALGERIA  ********.................
OTHER    *************************
```

Do not spend too much time studying the foregoing example before continuing.

Since *B* was obtained from *OIL* by summing over both months and years, it represents the total imports from each of the eight countries in the period covered. The result of the long expression above is clearly a barchart of these totals, labelled with the country names. We now examine the expression

```
      A,'.*'[1+B∘.≥(⌈/B)×(⍳N)÷N]
```

piece-by-piece, assigning names to intermediate results when convenient:

```
      N                           The width of the barchart
25
      Q←(⍳N)÷N                    Numbers from 0 to 1 in 25 equal steps
                                            (display if desired)
      ⌈/B                         The largest value to be charted
      C←(⌈/B)×Q                   Numbers from 0 to the largest value to be charted
```

NOTES

```
      S←B∘.≥C                          Comparison of each value of B with
                                       each value in the range to be charted
      S
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

      3 21↑1+S                         Examine a piece of 1+S
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1

      '.*'[1+S]
********************.....
******************.......
***************..........
*********................
********.................
********.................
********.................
*************************

      Z←'BARCHART◇ ''.*''[1+ω∘.≥((⍳α)÷α)×⌈/ω]'      The quote character must be
                                                     doubled
      Z
BARCHART◇ '.*'[1+ω∘.≥((⍳α)÷α)×⌈/ω]

      DEFINE Z

      NAMES,' ',40 BARCHART B

      )SAVE
```

NOTES

11c

Reading Expressions

```
      +/2 3 5
10
```

If no name is assigned to a result, it is displayed

```
      X←+/2 3 5
```

If a name is assigned, it is not displayed

```
      ⎕←A←+/2 3 5
10
```

The notation ⎕← causes the result to be printed regardless of name assignments. ⎕ does not denote a normal variable but serves to produce explicit display

Revise the function *BARCHART* to insert ⎕← just before (⍳α) and observe the behavior of Z←30 *BARCHART BX*

Revise *BARCHART* again to display a different intermediate result.

Revise *BARCHART* again to display all intermediate results.

NOTES

12a

Computations are often based on published rate tables. For example, a tax calculation function used in the form *TAX TI*, where *TI* is the taxable income, must use tax table data, such as the values that determine the tax brackets, and the rates applicable in each bracket. To avoid explicit concern with them when using the function *TAX*, they may be treated as **parameters,** that is, as named quantities used in the calculation.

The parameters used in tax calculations might be called *BP* (for Bracket Points), *BR* (for Bracket Rates in percent), and *BV* (for the value of the tax owing at the beginning of a bracket). They may be entered from a tax table such as shown in Table 2 at the end of this session. Thus:

```
      BP←1000×0 1 2 3 4 8 12 16 20 24 28 32 36 40 44 52
      BP←BP,1000×64 76 88 100 120 140 160 180 200
      BP                                 Compare BP with Table 2
```

If mistyped,
correct by using apl
←BP 2○
←(X↑BP),36,X↓BP

```
      .001×BPI←1 DIFF BP                 Examine intervals in BP
1 1 1 1 4 4 4 4 4 4 4 4 4 4 8 12 12 12 12 20 20 20 20 20

      0,+\BPI                            Recompute BP from intervals

      BR←14 15 16 17 19 22 25 28 32 36   Enter bracket rates
      BR←BR,39 42 45 48 50 53 55 58 60 62 64 66 68 69 70
      1 DIFF BR                          Examine intervals between rates
1 1 1 2 3 3 3 4 4 3 3 3 3 2 3 2 3 2 2 2 2 2 1 1

      BV←0,+\.01×(¯1↓BR)×1 DIFF BP       Compute bracket values and compare
                                         with table

      TI←42500                           Taxable income
      K←+/TI>BP
      K
14                                       Index of relevant bracket point
      BP[K],BR[K],BV[K]                  Relevant bracket quantities
40000 48 12140
      BV[K]+.01×BR[K]×TI-BP[K]           Tax calculation
13340
```

NOTES

```
      DEFINE  'TAX◇ BV[K]+.01×BR[K]×ω-BP[K←+/ω>BP]'

      TAX TI
13340
      TAX 82345
34700.1
```

Only the taxable income appears in these expressions; the **parameters** *BP*, *BR*, and *BV* **do not**, although they are used within the function

```
      DEFINE 'AVTR◇ 100×(TAX ω)÷ω'
      AVTR TI
31.38823529
      ROUND AVTR 82345
42
```

Average tax rate

Corporate taxes use different values of the parameters *BP*, *BR*, and *BV* as given in Table 1. Since the personal tax values might be needed again, retain them under the names *OBP* (old *BP*), etc.:

```
      OBP←BP
      OBR←BR
      OBV←BV
      BP←0 25000
      BR←22 48
      BV←0 5500
      TAX TI
13900
      ROUND (AVTR TI), AVTR 82345
33 40
```

The personal tax table parameters can be collected into a single "personal tax table" as follows:

```
      PTT←(3,ρOBP)ρOBP,OBR,OBV

      3 5↑PTT
  0 1000 2000 3000 4000
 14   15   16   17   19
  0  140  290  450  620
```

12c

Compare the following calculations with those of a previous page:

```
      TI←42500
      K←+/TI>PTT[1;]

      COL←PTT[;K]
      COL
40000 48 12140

      COL[3]+.01×COL[2]×TI-COL[1]

      COL[3]+.01×COL[2]×TI-(COL←PTT[;+/TI>PTT[1;]])[1]
```

* Define a general tax function *GTAX* such that *PTT GTAX TI* yields the correct tax on taxable income *TI* for the tax table *PTT*. Since the tax table information is now provided by an explicit argument, the function *GTAX* does not use any parameters.

* Specify a corporate tax table *CTT* and use it with the function *GTAX* to compute corporate taxes.

* Revise your summary of notation to bring it up to date.

```
      )SAVE
```

Taxable Income over:	But not over:	Tax equals	of excess over:
0	25,000	0+22%	0
25,000	......	5,500+48%	25,000

CORPORATE TAX TABLE
Table 1

Summary Lecture 9-12, slides 28-32

Taxable Income over:	But not over:	Tax equals	of excess over:
0	1,000	0+14%	0
1,000	2,000	140+15%	1,000
2,000	3,000	290+16%	2,000
3,000	4,000	450+17%	3,000
4,000	8,000	620+19%	4,000
8,000	12,000	1,380+22%	8,000
12,000	16,000	2,260+25%	12,000
16,000	20,000	3,260+28%	16,000
20,000	24,000	4,380+32%	20,000
24,000	28,000	5,660+36%	24,000
28,000	32,000	7,100+39%	28,000
32,000	36,000	8,660+42%	32,000
36,000	40,000	10,340+45%	36,000
40,000	44,000	12,140+48%	40,000
44,000	52,000	14,060+50%	44,000
52,000	64,000	18,060+53%	52,000
64,000	76,000	24,420+55%	64,000
76,000	88,000	31,020+58%	76,000
88,000	100,000	37,980+60%	88,000
100,000	120,000	45,180+62%	100,000
120,000	140,000	57,580+64%	120,000
140,000	160,000	70,380+66%	140,000
160,000	180,000	83,580+68%	160,000
180,000	200,000	97,180+69%	180,000
200,000	....	110,980+70%	200,000

PERSONAL TAX TABLE
Table 2

Summary Lecture 9-12, slides 28-32

13a

```
      A←⍞
R.V. JONES
      ⍴A
10
```

The boxed-quote denotes the literal list next entered on the keyboard

```
      DEFINE 'REVERSE◇⌽⍞'
      B←REVERSE
SLAP
      B
PALS
      SHOW ⍞
AT
AT◇ (1+ω÷100)*α
```

It can be used as an argument to *SHOW* to avoid using explicit quotes

```
      DEFINE 'SH◇ SHOW ⍞'
      SH
BEELINE
BEELINE◇ SQUAREROOT (α*2)+(ω*2)
```

Defining *SH* to take literal input makes it more convenient to use than *SHOW*

```
      DEFINE 'DEF◇ DEFINE ⍞'
      DEF
PLUS◇ α+ω
      3 PLUS 4
7
```

Similarly for *DEFINE*

* Use the function *DEF* to define a function *REV* which is related to the function *REVISE* as *DEF* is to *DEFINE*. Use the resulting function *REV* to revise the function *SQUAREROOT* so as to shorten its name.

NOTES

13a

Literal Input

Numeric Input

```
      X←⍎⍞
31 2 15
      X
31 2 15
      2×X
62 4 30
```

Execution of literal input yields numeric results

```
      DEF
KE◇ ⍎⍞
```

Define **and use** a Keyboard Entry function which produces numeric results from keyboard entries

*   Use the function *REV* to revise the function *TAX* to the following form

```
      KETAX◇ KE[K]+.01×KE[K]×ω-KE[K←+/ω>KE]
```

*   Use the function *KETAX* to perform the following pair of experiments:

```
      Q←KETAX 42500                    Q←KETAX 42500
0 25000                          OBP
0 25000                          OBP
22 48                            OBR
0 5500                           OBV

      Q                                Q
13900                            13340
```

numbers or variables

14a

Review the use of ⍞← at the end of Session 11 before continuing with the rest of this session.

```
      DEF
PR◇ ⍎⍞,0↑⍞←'ENTER ',ω                A Prompting function

      MSG←'DAYS IN CURRENT MONTH:'

      Z←PR MSG
ENTER DAYS IN CURRENT MONTH:
29
      Z
29
```

try deleting the '0↑', see what happens
↑
idiom

*   Use the function *REV* to revise the function *TAX* to the following form:

```
PRTAX◇ (PR 'BV')[K]+.01×(PR 'BR')[K]×ω-X[K←+/ω>X←PR 'BP']
```

*   Perform the following pair of experiments with the function *PRTAX*:

```
      PRTAX 42500                          PRTAX 42500
ENTER BP                             ENTER BP
0 25000                              OBP
ENTER BR                             ENTER BR
22 48                                OBR
ENTER BV                             ENTER BV
0 5500                               OBV
13900                                13340
```

*   Review the techniques for reading expressions and definitions presented in Session 11, and use them to analyze the details of the function *PRTAX*.

NOTES

14a

Prompting Functions

15a Data in your active workspace is available to you only, but can be made available to others by transferring it to a **file**. A file in **SHARP APL** is a collection of numbered components, each of which can contain a single variable. This may not seem like much information per component, but when combined with the ability to assemble several variables and functions into a **package**, files can be very useful indeed. Packages will be explored in the next Session.

Notes

Before you can use a file, you must first create it. You do this with the first of several system functions which SHARP APL uses to manipulate files. Each of these functions has a name beginning with the quad character, to distinguish these functions from the ones you create.

```
      'TAXES' ⎕CREATE 1
```

Create a file named *TAXES* and **tie** it to the number 1. You will be using this number to refer to the file with other system file functions

```
      OBP ⎕APPENDR 1
1
      OBR ⎕APPENDR 1
2
      OBV ⎕APPENDR 1
3
```

Add *OBP* as a component of the file by appending it to the file (the 1 is the tie number). The result is the number of the component you added

```
      A←⎕READ 1 2
```

Individual items are retrieved by reading a component

```
      A
14 15 16 17 19 22 25 28 32 36 39 42 45 48 50 etc.
```

15a

Public Data (Files)

Notes

```
      □SIZE 1                         How big is the file so far?  See the
1 4 2000 10000                        SHARP APL Reference Manual
      'STRING1' □APPENDR 1            for the meaning of the result
4
      □READ 1 4
STRING1
      'STRING TWO' □REPLACE 1 4       You can change the data in any
      □READ 1 4                       existing component
STRING TWO
```

Files may contain hundreds of millions of characters of data, and are sometimes used to complement the storage available in the active workspace, which, in the present system, is limited to about 650 thousand.

```
      1↑□AI                           Current active account number

      □LIB 1↑□AI                      File library of active account

      □NAMES                          Names of files you have tied, with
  1234567 TAXES                       the account numbers which own them
      □NUMS                           Tie numbers for those file
1
      □UNTIE 1                        Release the file
```

Note: Files which have already been created may be tied either with *□TIE*, which gives you exclusive access to the file until you *□UNTIE*, or with *□STIE*, which allows others to *□STIE* the file at the same time. Unless you have pressing need for exclusive access to a file, you should use *□STIE*.

16a Several different APL objects - both functions and variables - may be bundled together into a special type of variable called a **package**. One common use of packages is to group related functions and data for storage together in a single component of an APL file. Then all these objects may be retrieved in a single operation.

Notes

The **SHARP APL** language includes several system functions for manipulating packages.

```
      TAXPACK←⎕PACK 'PRTAX PR BP BV BR OBP OBV OBR trap'
      TAXPACK
**PACKAGE**
```

You have packaged up the tax functions and both sets of tax tables plus *trap* (or *TRAP* on some terminals), whose secret use you'll see in the next Session. You can't see what is in a package 'from the outside'. You must use special functions to open it up.

```
      ⎕PNAMES TAXPACK                    What objects are in TAXPACK
PRTAX
PR
BP
BV
BR
OBP
OBV
OBR
trap
      'BP' ⎕PVAL TAXPACK                 What is the value of the object BP
0 25000
      )SAVE                              Save the workspace before going on!
```

16a

Packages

```
      'TAXES' ⎕STIE 1
      TAXPACK ⎕REPLACE 1 4
```

Place the entire package in one component

```
      )CLEAR
CLEAR WS
      TP←⎕READ 1 4
      ⎕PDEF TP
```

Read the package back in
Now unpack it into the workspace

```
      )FNS
PR PRTAX
      )VARS
BP BR BV OBP OBR OBV TP trap
```

Try using *PRTAX* (see Session 14). Note that you could have skipped the step of creating the variable *TP* by unpacking the component when you read it.

```
      ⎕PDEF ⎕READ 1 4
```

You can also add objects to an existing package.

```
      DESCRIBE←'THIS PACKAGE CONTAINS TAX FUNCTIONS AND TABLES'
      BIGGERPACK←TP ⎕PINS ⎕PACK 'DESCRIBE'
```

NOTE: *⎕PINS* will replace objects in the 'target' package with the same names as objects in the 'source' package.

* See the **SHARP APL Reference Manual** for more information on packages and the system functions which manipulate them.

17a

To define functions which perform sequences of actions, we will use an alternative form of definition called the **canonical**, or **del** form.

Notes

```
        ∇INQUIRY
[1]     '    ENTER FILE NAME'
[2]     A←⍞
[3]     '    ENTER COMPONENT NUMBER'
[4]     B←⍞
[5]     A ⎕STIE 1
[6]     ⎕READ 1,B
[7]     ⎕UNTIE 1
[8]     ∇
```

The bracketed line numbers are printed automatically

Close the definition

```
        INQUIRY
   ENTER FILE NAME
TAXES
   ENTER COMPONENT NUMBER
2
14 15 16 17 19 22 25 28 32 etc.
```

Use the function

```
        ∇INQUIRY
[8]     →1
[9]     ∇
```

Reopen the definition and add a branch on line 8 which will cause line 1 to be executed next.

```
        INQUIRY
   ENTER FILE NAME
TAXES
   ENTER COMPONENT NUMBER
```

17a

Sequential Functions

Notes

```
1
0 1000 2000 3000 4000 8000 etc.
   ENTER FILE NAME                           Repetition of line 1 begins here
TAXES
   ENTER COMPONENT NUMBER
2
14 15 16 17 19 22 etc.
   ENTER FILE NAME                           Execution of INQUIRY continues.
T                                            Enter O backspace U backspace T to
INPUT INTERRUPT                              escape from endless loop.
INQUIRY[2]   A←⍞
                  ∧
      →                                      Branch out of halted function.

      ∇INQUIRY[8]→3∇                         Make branch go to line 3 instead of 1

      INQUIRY
   ENTER FILE NAME
TAXES
   ENTER COMPONENT NUMBER
1
0 1000 2000 3000 4000 8000 etc.
   ENTER COMPONENT NUMBER
2
14 15 16 17 19 22 etc.
   ENTER COMPONENT NUMBER
T                                            Enter O backspace U backspace T
INPUT INTERRUPT
```

INQUIRY[~~2~~4] ~~*A*~~*B*←⍞

```
                    ∧
```

Notes

```
      →
```

Branch out of halted function.

```
      ∇INQUIRY[4.5] →(0=⍴B)/0
```

Add a conditional branch between lines 4 and 5 to branch to 0 (i.e., terminate) if the component number entered was empty (i.e., a return only)

* Use the function *INQUIRY* and verify that an empty entry will terminate the loop.

```
      ∇INQUIRY[⎕]∇
```

Display the modified definition

```
      ∇INQUIRY
[1]   '   ENTER FILE NAME'
[2]   A←⍞
[3]   '   ENTER COMPONENT NUMBER'
[4]   B←⍞
[5]   →(0=⍴B)/0
[6]   A ⎕STIE 1
[7]   ⎕READ 1,B
[8]   ⎕UNTIE 1
[9]   →3
     ∇
```

You have now learned two modes of function definition.

```
      DEFINE 'F◇ α+α÷ω'
```

This mode is called **direct** definition

```
      2 + 3 F 4
5.75
      ∇Z←X G Y
[1]   Z←X+X÷Y
[2]   ∇

      2+3 G 4
5.75
      M←⎕CR 'G'
      M
Z←X G Y
Z←X+X÷Y

      ⎕EX 'G'
1
      3 G 4
SYNTAX ERROR
      3 G 4
        ∧
      ⎕FX M
G
      3 G 4
3.75
      ⎕CR 'F'
r←α F ω;⎕TRAP
⎕TRAP←trap
r←α+α÷ω
⍝F⋄ α+α÷ω
```

This mode is called **canonical** function definition.

The functions *F* and *G* are equivalent. ⎕*CR* produces the canonical representation of its argument

Expunge *G*

⎕*FX* fixes the definition of the function represented by its argument

Though produced by direct definition *F* is actually represented in canonical form, with a final **comment line** (which is never actually executed) for use by the function SHOW, and with an initial **trap** line which prevents suspension of the function in execution (now you know where *trap* is used).

* Use ⎕*CR* to display the canonical definitions of other functions (such as *TAX*, *PRTAX*, etc.) which you have previously defined.

* The function *EDIT* used in the function *REVISE* defined in Session 8 provides for convenient editing of functions defined in the direct definition form. Corresponding facilities for editing the del form may be found in the manuals and (for full screen terminals) in workspace 7 *DEL*.

15a

Data in your active workspace is available to you only, but can be made available to others by transferring it to a file. The following sequence uses the function *TO* to transfer tax table data to a file called *TAXES*:

```
      OBP TO 'TAXES 1'                 The indices 1, 2, and 3 identify
                                       the individual items in the file
      OBR TO 'TAXES 2'

      OBV TO 'TAXES 3'

      A←GET 'TAXES 2'                  Individual items are retrieved by index
      A
14 15 16 17 19  22 25 28 32 36 39 42 45 48 50 53 55 58 etc.

      TABLE←GET 'TAXES 1 2 3'          A set of items may be retrieved together
      ⍴TABLE                           This is the same combined table used
3 25                                   at the end of Session 10

      GET 'TAXES 3 2'                  Items may be selected in any order
```

If the first dimension of an array (list, table, or list of tables) has the value *N*, (for example, 1↑⍴*OIL* is 7), then it may be distributed to *N* items of a file by a single operation. For example:

```
      OIL TO 'IMPORTS 72 73 74 75 76 77 78'
```

*   Get individual items from the *IMPORTS* file to verify the effect of the preceding expression.

```
      OIL TO 'IMPORTS 100'             The whole array may be entered
                                       as a single item

      NAMES TO 'IMPORTS 1'             Non-numeric data may be entered
```

NOTES

The functions *RANGE* and *REMOVE* are useful in managing files:

```
      RANGE 'IMPORTS'                Gives range of indices
72 73 74 75 76 77 78 100 1

      REMOVE  'IMPORTS 100'          Removes item 100

      RANGE 'IMPORTS'
72 73 74 75 76 77 78 1

      DEFINE 'SORT ◇ ω[⍋ω]'

      SORT RANGE 'IMPORTS'
1 72 73 74 75 76 77 78
```

Files may contain hundreds of millions of characters of data, and are sometimes used to complement the storage available in the active workspace, which, in the present system, is limited to about one-quarter million.

```
      1↑⎕AI                          Current active account number

      FLIB 1↑⎕AI                     File library of active account
```

* Data bases employ files. To get an idea of how data bases are used, obtain a listing of those available on the system you are working on, and perhaps try one of interest. On the Sharp APL system you may try the following:

skip if needed.

a) First save your current workspace, then load 1 *DATABASES*, then enter *DATABASES*, and then enter answers to the prompts provided. In particular, answer the first with 2 and the second with *ALL* to get a complete listing.

b) Load 11 *BIBLIOGRAPHY*, then enter *START* and answer the successive prompts with *LR* (for List Records) and *AUTHOR=FALKOFF* and *TITLE,SOURCE* and *DATE* and finally with a single space. When printing is completed, enter *QUIT*.

NOTES

Should say)LOAD MINE saved earlier

16a

To define functions which perform sequences of actions, we will use an alternative form of definition called the **canonical**, or **del** form.

```
      ∇INQUIRY
[1]   '   ENTER FILE NAME'          The bracketed line numbers are
[2]   A←⍞                           printed automatically
[3]   '   ENTER FILE INDEX'
[4]   B←⍞
[5]   GET A,' ',B
[6]   ∇                             Close the definition

      INQUIRY                       Use the function
   ENTER FILE NAME
TAXES
   ENTER FILE INDEX
2
14 15 16 17 19 22  25 28 32 36 39 42 45 48 50 53 55 58 etc.

      ∇INQUIRY                      Reopen definition and add a branch on
[6]   →1                            line 6 which will cause line 1
[7]   ∇                             to be executed next
                                      Berry p.179,180 Branching
      INQUIRY
   ENTER FILE NAME
TAXES
   ENTER FILE INDEX
1
0 1000 2000 3000 4000 8000 etc.
   ENTER FILE NAME                  Repetition of line 1 begins here
TAXES
   ENTER FILE INDEX
2
14 15 16 17 19 22 etc.
```

NOTES

Berry p.63
function definition discussion

IBM p.81
function definition discussion.

16a
Sequential Functions

```
   ENTER FILE NAME
Ø
INPUT INTERRUPT
INQUIRY[2]  A←⍞
              ∧
```

Execution of *INQUIRY* continues
Enter *O* backspace *U* backspace *T* to escape from endless loop

```
      →
```

Branch out of halted function

```
      ∇INQUIRY[6]→3∇
```

Make the branch go to line 3 instead of 1 to avoid reentry of the file name

```
      INQUIRY
   ENTER FILE NAME
TAXES
      ENTER FILE INDEX
1
0 1000 2000 3000 4000 8000 etc.
   ENTER FILE INDEX
2
14 15 16 17 19 22 etc.
   ENTER FILE INDEX
Ø
INPUT INTERRUPT
INQUIRY[4] B←⍞
             ∧
```

Enter *O* backspace *U* backspace *T*

```
      ∇INQUIRY[4.5] →(0=⍴B)/0 ∇
```

Add a conditional branch between lines 4 and 5 to branch to 0 (i.e., terminate) if the index entered was empty (i.e., a return only)

* Use the function *INQUIRY* and verify that an empty entry will terminate the loop.

```
      ∇INQUIRY[⎕]∇
    ∇ INQUIRY
[1]   '    ENTER FILE NAME'
[2]   A←⍞
[3]   '    ENTER FILE INDEX'
[4]   B←⍞
[5]   →(0=⍴B)/0
[6]   GET A,' ',B
[7]   →3
    ∇
```

Display the modified definition

*   The function *EDIT* used in the function *REVISE* defined in Session 8 provides for convenient editing of functions defined in the direct definition form. Corresponding facilities for editing the del form may be found in the manuals.

*   Revise your summary of notation.

Summary Lecture 13-16, slides 33-37

NOTES

```
      DEFINE 'F◇ α+α÷ω'
```

This mode of function definiton is called **direct** definition

```
      2 + 3 F 4
5.75
      ∇Z←X G Y
[1]   Z←X+X÷Y
[2]   ∇
```

This mode is called **canonical** function definition

```
      2+3 G 4
5.75
```

The functions *F* and *G* are equivalent

```
      M←⎕CR 'G'
      M
Z←X G Y
Z←X+X÷Y
```

⎕*CR* produces the canonical representation of its argument

```
      ⎕EX 'G'
1
```

Expunge *G*

```
      3 G 4
SYNTAX ERROR
      3 G 4
        ^
```

I get
VALUE ERROR

```
      ⎕FX M
G
      3 G 4
3.75
```

⎕*FX* fixes the definition of the function represented by its argument

```
      ⎕CR 'F'
R←α F ω;⎕TRAP
⎕TRAP←TRAP
R←α+α÷ω
⍝F◇ α+α÷ω
```

Although produced by direct definition, *F* is actually represented in canonical form, with a final **comment line** (which is never actually executed) for use by the function *SHOW*, and with an initial **trap** line which prevents suspension of the function in execution (This result may differ on different APL systems.)

* Use ⎕*CR* to display the canonical definitions of other functions (such as *TAX*, *PRTAX*, etc.) which you have previously defined.

NOTES

17a

Modes of Definition

18a

NOTES

In a **conditional** definition, the function is defined by three expressions separated by diamonds. The first expression is a **proposition** (having only results 0 or 1, which may be thought of as **false** or **true**) which is executed first; if it is false, the next expression is executed, otherwise the last is executed. For example:

```
      DEF
ABC◇ ω>0 ◇ ω*2 ◇ ω*.5

      ABC 4
2
      ABC ¯4
16
```

Square root if argument is positive, square otherwise

* The function *SHOW* is conditional. Display it and analyze its definition, using the techniques of stepwise examination presented in Session 11.

In a **recursive** definition, the function being defined recurs in its own definition. For example:

```
      DEF
FAC◇ ω=0 ◇ ω×FAC ω-1 ◇ 1

      FAC 4
24
```

probably will have to work through this w/ students

Reading from the definition we see that *FAC* 4 is equivalent to 4×*FAC* 3, that *FAC* 3 is equivalent to 3×*FAC* 2, etc. Thus, the following expressions are all equivalent:

```
      FAC 4

      4×FAC 3

      4×3×FAC 2

      4×3×2×FAC 1

      4×3×2×1×FAC 0
```

Finally, since the proposition ω=0 is true for the case *FAC* 0, the result of *FAC* 0 is the rightmost expression in the definition, namely 1. Thus:

```
      FAC 4 ↔ 4×3×2×1×1
```

* Analyze the recursive definition of the function *EDIT*.

* The function *FIB* applied to a positive whole number *N* produces the first *N* **fibonacci** numbers, each of which is the sum of the preceding two. It is defined by *FIB*◇ ω=1◇Z,+/¯2↑Z←*FIB*ω-1◇1. Use and analyze it.

ratio of volume enclosed in each chamber of chambered nautilus.

19a

A brief introductory APL course cannot usefully treat all primitives in the language, and even many of those used cannot be treated in full generality. The student should therefore become familiar with one or more APL reference manuals, and should be on the alert for further primitives which may simplify the work of programming. The following examples suggest a few of the primitives which should be explored:

NOTES

Format

```
      A←3 4↑OIL72÷1000
      A
 6.456  3.082  6.474  6.88
 6.384  5.806  4.335  6.069
26.915 25.426 26.923 25.33
```

```
      2⍕A
  6.46  3.08  6.47  6.88
  6.38  5.81  4.34  6.07
 26.92 25.43 26.92 25.33
```

A one-element left argument determines the number of digits following the decimal point

```
      8 2 ⍕ A
   6.46    3.08    6.47    6.88
   6.38    5.81    4.34    6.07
  26.92   25.43   26.92   25.33
```

Two elements specify column width and decimal point

```
      5 0 8 2 5 0 8 2 ⍕ A
```

Two elements per column specify widths and decimal points for **each** column

Inner Products

```
         V←1 2 3 4 5
         W←2 0 1 3 2
         +/V×W
27
         V+.×W
27
         M←4 5ρι20
         M
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20

         M+.×W
27 67 107 147

         3 0 1 2+.×M
46 52 58 64 70

         V+.×V≠3
12
         M⌈.×W
12 27 42 57
         M∧.=11 12 13 14 15
0 0 1 0
```

Berry 191 →

The manuals provide illuminating examples and diagrams of inner products

NOTES

Transpose

The transpose function changes the order of the axes of its right argument. Thus:

`OIL`	Years by suppliers by months
`2 1 3 ⍉ OIL`	Suppliers by years by months
`1 3 2 ⍉ OIL`	Years by months by suppliers
`⍉ OIL`	Months by suppliers by years

Berry 144 - 148

* Examine the shapes of each of the foregoing results.

Selector Generators

Read the brief section on this topic in the IBM Manual, and experiment with some of the examples given there. Also use the **atomic vector** ⎕*AV* as a left argument of the **index of** function in expressions such as ⎕*AV*⍳'*ABCD*' and ⎕*AV*⍳'*A̲B̲C̲D̲*' and ⎕*AV*⍳'+-×'.

Latent Expression

Read ~~the discussion of this topic in the manuals and perform appropriate experiments.~~

```
      DEFINE ⍞
SET◇ ⍎ω,'←⍎⍞',0↑⎕←'ENTER ',ω
      DEFINE ⍞
SETUP◇ 0=1↑ρω◇SETUP (1 0,0ρSET ω[1;])↓ω◇''
      NAMES←3 2ρ'BPBRBV'        Make a table of relevant names
      NAMES
BP
BR
BV
      ⎕EX NAMES                 Expunge these names to show that the
1 1 1                             setting produced by SETUP is effective
      BP
VALUE ERROR
      BP
      ∧
      SETUP NAMES
ENTER BP
0 25000
ENTER BR
22 48
ENTER BV
0 5500
      BP
0 25000
      TAX TI
13900
```

* Repeat the setup of names, using *OBP*, *OBR*, and *OBV* as responses. Then enter *TAX TI*. (The result should be 13340.)

* Use the techniques of Session 11 to determine the behaviour of the following functions, and then state clearly what they do:

```
ST◇0=α◇⍎ω,'←⍎⍞',0ρ⎕←'ENTER ',ω◇⍎ω,'←⍎⍞'

SP◇0=1↑ρω◇αSP(1 0,0ραST ω[1;])↓ω◇''
```

Too little info??

Summary Lecture 17-20, Slides 38,39

NOTES

20a

Presetting Parameters

SUPPLEMENTARY EXERCISES

The first part of each exercise number indicates the session with which it is associated.

NOTES

1.1 Annotate each of the following expressions in the manner shown in Session 1:

```
KM←MILES×1.6

MILES←KM÷1.6

GRAMS←POUNDS÷.0022

F←32+(1.8×C)

C←(F-32)÷1.8
```

1.2 For each of the following expressions, first enter expressions which assign values to the arguments (i.e. the names) used, and then enter the given expression:

```
PRICE×QUANTITY

FORCE×DISTANCE

LENGTH×WIDTH

COST←PRICE×QUANTITY

WORK←FORCE×DISTANCE
```

1.3 Annotate the sequence produced in Exercise 1.2. For example, the final expression might be annotated by "The work done is computed as the force applied times the distance through which it acts".

1.4 The names used for the arguments and results in Exercise 1.2 are mnemonic in that they suggest the interpretation to be given to the expressions in which they appear. However, all of the expressions concern the same function (multiplication) and their similarity could be emphasized by using brief neutral names (as in *Z*←*X*×*Y*) and providing annotations for each area of application, accounting , physics, etc. For each of the following expressions suggest two or more areas of application and provide annotations for them:

```
Z←X÷Y

Z←(A+B)÷2

C←A⋆B

Z←(⌊A×B)÷B
```

2.1 Use the function ⍳ in expressions to:

a) assign to *MILES* the integer values from 0 to 10, inclusive.

b) assign to *F* the integer values from 0 to ¯40.

c) assign to *POUNDS* the integer and half-integer values from 5 to 15.

2.2 Use the expressions from Exercise 1.1 to obtain metric and Celsius equivalents of the quantities entered in Exercise 2.1 above.

2.3 Experiment with revising an entry (before striking the carriage return or entry key) by backspacing to some point, striking the **attention**, **linefeed**, or **index** key (according to the particular terminal) and then continuing with the revised entry.

2.4 Annotate the following expressions:

```
MILES,(MILES×1.6)

2,⍴MILES

(2,⍴MILES)⍴(MILES,(MILES×1.6))
```

2.5 Consult the definition of the function ÷ in the Table of Primitive Scalar Functions in an APL reference manual, and check your understanding of it by performing experiments with both one argument (as in `÷4`) and two (as in `3÷4`). Repeat for a few other functions in the table.

3.1 Determine the behaviour of the function denoted by `?` by applying it in expressions such as `?6` and `?6 6` and `?20 20 20`. Confirm your understanding of it by consulting the manual, and then annotate the following expressions:

```
NINES←3 5 12 ⍴ 9

RAIN←?NINES
```

3.2 Enter each of the following expressions, examine the results, and annotate them in the manner shown for the first, assuming that *RAIN* represents the monthly rainfall in three succeeding years in five counties:

```
+/[1] RAIN   Three-year rainfall totals from each county by months

+/[2] RAIN

(+/[1] RAIN)÷3

(+/[2]RAIN)÷5
```

NOTES

3.3 Write expressions to determine:

a) the average yearly rainfall for each county for each year.

b) the average yearly rainfall for the entire area over the three years recorded.

4.1 a) Carefully observing the number of spaces within the quotes, enter the expression:

```
Z←3 8 ρ'JANUARY FEBRUARYMARCH   '
```

b) Examine the value of *Z* and use the result as a guide for an expression which assigns to *TM* the names of all the months spelled out in full.

c) Repeat part b using an expression of the form *TM*←*MONTHS*,(12 6ρ*X*), where *MONTHS* is the table of abbreviated spellings produced in Session 4, and where you have previously assigned to *X* the characters necessary to complete the spelling of the months.

4.2 Experiment with the function ⍉ by applying it to each of the tables *MONTHS* and *TM*, and confirm your understanding of its behaviour by consulting the manual.

4.3 Apply the function ⍉ to various numerical tables produced in earlier sessions.

4.4 If at this point you wish to experiment with producing "reports" from both numerical data and tables of literal characters, perform the experiments shown at the beginning of Session 9, and consult the manual for the definitions of the functions ⍕ and ⍎.

5.1 Read the introductory material on workspaces and libraries in the APL language manual, and confirm your understanding of it by experiments.

6.1 "The effect of executing the expression *ROUND X* for any argument *X* is the same as would be obtained by substituting *X* for each occurrence of the symbol ω in the expression which defines the function *ROUND*, that is, in the expression `⌊.5+ω`". Confirm this statement by entering expressions using different values of the argument *X*.

6.2 Make a statement similar to that in Exercise 6.1 concerning the function *AT* which applies to two arguments. Confirm your statement as in Exercise 6.1.

6.3 Examine the expressions used in earlier sessions and identify (and verify by experiment) those in which some or all of the parentheses could be omitted without changing their meanings.

7.1 For each of the following functions, describe its purpose and illustrate its use by entering its definition and applying it to appropriate arguments:

a) `ROWMAX◇ ⌈/[2]ω`

b) `PLANEMIN◇ ⌊/[1]ω`

c) `MAX◇ ⌈/[α]ω`

7.2 The following functions concern areas and other properties of triangles, rectangles, and other polygons. In each case define the function and choose appropriate arguments to illustrate its use.

a) `PERIMETER◇ +/ω`

b) `SEMIP◇ .5×PERIMETER ω`

c) `TRIAREA◇ (×/(SEMIP ω)-0,ω)*.5`

NOTES

d) *DIAGONAL*◇ (+/ω*2)*.5

e) *TEST*◇ ∧/ω≤*SEMIP* ω

Hint: *TEST* 2 5 4 yields 1, showing that the lengths 2 5 4 can form the sides of a triangle. Lengths 2 7 4 cannot.

7.3 The following functions concern boxes, i.e. rectangular solids. Define each function and apply it to appropriate arguments.

a) *VOL*◇ ×/ω

b) *PERIM*◇ 2×+/ω

Perimeter with cubes??

c) *SURFACE*◇ 2×+/ω×1⌽ω

d) *DIAGONAL*◇ (+/ω*2)*.5

8.1 a) Using the formula "four-thirds pi times *R* cubed" as the formula for the volume of a sphere of radius *R*, and the value 3.1416 as an approximate value of pi, define a function *SVOL* for the volume of a sphere.

b) Revise the function *SVOL* to replace 3.1416 by the more exact value ○1.

9.1 Define the function *CUMSUMS*◇ α,⍕+\[2] ω, execute the expression *NAMES CUMSUMS OIL*73, and discuss the meaning of the result. Note that the backslash symbol \ has not been used before -- experiment with it or consult the manual.

9.2 Repeat Exercise 9.1 with the function *CUMMAX*◇ α,⍕⌈\ω.

NOTES

10.1 Discuss the uses of the following functions:

a) `FORYEARS◇ α[ω-1971;;]`

b) `FORCOUNTRIES◇ α[;ω;]`

c) `FORMONTHS◇ α[;;ω]`

10.2 State the meaning of each of the following results:

a) `7 8 6↑OIL`

b) `+/[3] 7 8 6↑OIL`

c) `0 0 6↓OIL`

d) `7 8 ¯6↑OIL`

e) `3 8 ¯6↑OIL`

f) `((6ρ1),6ρ0)/[3]OIL`

g) `(2|ι12)/[3]OIL`

h) `(1=3|ι12)/[3]OIL`

11.1 Use the techniques of Session 11 to read the following definitions:

a) `DIVISIBILITYTABLE◇ 0=(ιω)∘.|ιω`

b) `DT◇ 0=(ιω)∘.|ιω`

c) `PRIMES◇ (2=+/[1]DTω)/ιω`

d) `SQRT◇ ω*.5`

Supplementary Exercises

14.1 Enter and experiment with the functions *DRILL* and *COACH* as follows:

```
      DRILL◊ ⍎⍞,0⍴⎕←'MULTIPLY ',(⍕ω[1]),' AND ',⍕ω[2]

      Z←DRILL 3 7

MULTIPLY 3 AND 7
21
      Z
21

COACH◊ (2 5⍴'RIGHTWRONG')[1+(×/ω)≠DRILLω;]

     COACH 3 7
MULTIPLY 3 AND 7
20
WRONG
```

14.2 Experiment with the function

```
      TEST◊ (2 5⍴'RIGHTWRONG')[1+(×/X)≠⍎⍞,0⍴⎕←X←?α⍴ω;]
```

14.3 Revise the function *TEST* to deal with functions such as + and ⌈ instead of ×.

15.1 Because of differences in the file facilities provided by APL systems, Exercises 15.1-6 may be expected to apply only to a SHARP APL system.

Every file name is associated with the account number of a user that produced it, and the full reference to the file includes the account number. For example, on the SHARP APL system you may enter:

```
      CAPITALS←GET '13 NATIONS 1'

      COUNTRIES←GET '13 NATIONS 2'
```

to get items 1 and 2 of the file called *NATIONS* in account number 13.

(If the account number is not specified, as in `GET 'TAXES 2 1'` in Session 15, it is assumed to be the account number in use, that is, `1↑⎕AI`.)

15.2 a) Examine the shapes of the tables *CAPITALS*, and *COUNTRIES*, produced in Exercise 15.1.

b) Print the first 15 countries and then print the first 15 capitals.

c) Make a single table of the countries and capitals side by side, and print the last ten rows of it.

15.3 Define a function as follows:

```
      DEFINE ⍞

      NATIONS◇ GET '13 NATIONS ',⍕ω
```

Use expressions such as `AREAS←NATIONS 3` to assign the names *AREAS*, *POPULATION*, *LATITUDES*, and *LONGITUDES* to file elements 3, 4, 5, and 6, respectively. (Use shorter mnemonic names if you prefer).

15.4 a) Enter the expression `COUNTRIES[⍋AREAS;]` and state its meaning.

b) Define a function called *BY* such that the expression `COUNTRIES BY POPULATION` produces a table of the countries in order of increasing population.

NOTES

c) Produce a table of capitals in **decreasing** order on the populations of their countries.

d) Produce a table of countries in increasing order on population **density**.

15.5 Access to a file can be restricted to any desired list of **account numbers** (your own account number can be determined by entering 1↑⎕*AI*), and the **type** of access permitted (such as **read only**) can be specified for each account number. The general facilities provided for controlling file access are rather complex (as may be seen, for example, from the discussion of **file access control** in the SHARP APL manual), but the function *ACCESS* provides simple controls adequate for a wide range of use.

In order to observe the effects of such restrictions it will be necessary to attempt to access a file from one or more different account numbers. We will therefore assume that you have the use of two terminals other than your own which have the account numbers (as determined by entering 1↑⎕*AI* on each) *AN*1 and *AN*2. We will further assume that your own account number is *AN*.

a) Verify that the files that you have produced using the function *TO* permit reading by anyone, but writing only by your account.

b) The expression *A TO* '*TAXES* ¯1' not only enters *A* as a component of the file *TAXES* having the special index ¯1, but the value of *A* (called the **access control matrix**) will then control access to the file until such time as the component ¯1 may be changed by a further use of the function *TO*.

Enter *A*←3 2⍴*AN*,1,*AN*1,0,*AN*2,1 and *A TO* '*TAXES* ¯1'. Then display *A* and try to determine the significance of the access matrix by trying to read and write the file *TAXES* from the different account numbers.

c) The **access codes** in the second column of *A* specify the access permitted to the corresponding account numbers in the first column, the code 1 permitting complete access, and the code 0 permitting read access only.

Verify that (1 2⍴(1↑⎕*AI*),1) *TO* '*TAXES* ¯1' provides a private file.

d) The number `0` in the first (i.e., account number) column of the access control matrix denotes **all** accounts. Verify that `(1 2ρ0 1) TO 'TAXES ¯1'` gives complete access to all accounts.

15.6 The function `INDEX◇(α∧.=(¯1↑ρα)↑ω)⍳1`, can be used to determine the row index of an entry in a table. For example:

```
      ⎕←T←3 5ρ'ABEL BAKERDOG  '
ABEL
BAKER
DOG

      T INDEX 'DOG'
3
```

a) Use the information in Exercises 15.1-3 to construct a table *T* of names appropriate to the components 1 through 6 of the file `13 NATIONS`.

b) Define the function `G◇GET '13 NATIONS ',⍕T INDEX ω` and experiment with expressions of the form `G 'AREAS'`.

15.7 If you are using a non-SHARP APL system, study the file facilities available on your system and modify the functions *TO*, *GET*, *RANGE*, *REMOVE*, and *FLIB* so as to use the available file facilities, but to preserve as far as possible the appearance of the functions to the user. If possible, include access control as described in Exercise 15.5.

15.8 Read and experiment with the functions *PRINT* and *PRALL*.

16.1 Read the sections on function definition in both manuals. In particular, read about **labels** and experiment with their use.

17.1 Display the definition of the function *INQUIRY* by entering `⎕CR 'INQUIRY'`, and compare the display with that produced at the end of Session 16.

17.2 The function *PRTAX* defined in Session 14 assigns a value to the argument *X* for use within the definition. This name is made "local" to the function definition so that it does not affect the value of a similarily named variable. Use `⎕CR 'PRTAX'` to display the definition of the function, note the inclusion of the name *X* in the header, and re-read the discussion of local variables in the manuals.

17.3 Use `⎕CR` to display the definitions of other existing functions, and carefully compare the details of the del form of definition with the direct form given by the function *SHOW*, or by the comment line at the end of the del form.

17.4 As an exercise in the use of the canonical form of function definition, read the section **A Prepared Workspace** in the introduction to the IBM Manual, and enter and use some or all of the functions and variables shown.

Because of small but significant differences in the treatment of **bare output** (denoted by the phrase `⍞←`) on different systems, the function *NEWSTOCK* will not work on all APL systems. A study of **bare output** in the manuals will suggest ways to rectify the matter; the easiest (which works on all systems) is to simply omit all occurrences of the phrase.

18.1 Use the techniques of Session 11 to read the following function definitions:

a)
```
SORT◇ 0=ρω ◇ (U/ω),SORT (~U←ω=⌊/ω)/ω ◇ ι0
```

b)
```
TSORT◇ α[((ρω)[2]) S αιω]
S◇ α=0 ◇ (α-1) S ω[⍋ω[;α];] ◇ ω
```

Hint: Try `'ABCDEFGHIJKLMNOPQRSTUVWXYZ ' TSORT NAMES`

c) Sort into alphabetical order the table *CAPITALS* produced in Exercise 15.1.

18.2 In most published use of direct definition (as in all of the references given in this exercise) the separation of the segments is indicated by the colon (:) rather than the diamond. The order of the expressions also differs, with the **proposition** occurring between the other two expressions rather than before them.

Read the discussion of recursive definition on pages 141-146 of Iverson **Elementary Analysis** (APL Press, Palo Alto, CA, 1976), and do Exercises 10.16 to 10.23. Study the functions defined in the two papers by D.B. McIntyre, and the one paper by K.E. Iverson in the proceedings of **An APL Users Meeting**, I.P. Sharp Associates, 1978. Many recursive definitions may also be found in **Notation as a Tool of Thought**, CACM, Aug., 1980.

19.1 Consult the manuals for the defintions of format, inner product, and transpose, and try further experiments to confirm your understanding of them.

19.2 Display the matrix *Q*←(⌈(⍳12)÷3)∘.=⍳4 and show that the expression *OIL*+.×*Q* produces quarterly summaries of the oil data.

20.1 Use the techniques of Session 11 to examine the definition of the function *SETUP*.

20.2 Define a function *TOTAL* which prints the prompting message *ENTER YEAR DESIRED*, and then gives the overall total oil imports for the year entered.

20.3 Modify the function *TOTAL*, if necessary, to allow the entry of several years (e.g., 73 75 77), and compute the single overall total for those years. Ensure that the function still works for a single year.

NOTES

20.4 Define a function *MAXCHANGE* which gives, for each supplier, the maximum change (either positive or negative) occurring from year to year in total yearly imports. The argument to which this function must apply is a two-element list specifying the first and last years in the desired range. For example, *MAXCHANGE* 73 77 should treat years 73 to 77, inclusive.

20.5 Define a function *IMSS* which yields the index of the most stable supplier for the range of years specified by its argument, that is, the index of the supplier whose yearly totals showed the least maximum change as given by *MAXCHANGE*.

20.6 Define a function *NMSS* which yields the name of the most stable supplier.

20.7 Choose computational procedures of interest in your particular discipline (consulting handbooks or textbooks if necessary), and define and test corresponding functions. For example, the text **Essentials of Managerial Finance** (by J.F. Weston and E.F. Brigham, published by The Dryden Press, Hinsdale, Ill.) gives, on pages 180-182 of the third edition, a procedure for computing **economic order quantity**, and an example of its use. The statement of the procedure may be paraphrased as follows:

> The economic order quantity is the square root of twice the product of the annual sales in units with the quotient obtained by dividing the fixed cost of placing and receiving an order by the carrying cost per unit of inventory.

Weston and Brigham quote an example in which the annual sales are 100 units, the carrying cost is $.20 per unit, and the fixed cost of ordering is $10.00; they give the correct result of the economic order quantity computation as 100 units.

a) Choose names for the relevant variables, and assign to them the values specified in the example.

b) Write the expression for the economic order quantity, and verify that the result agrees with that given by the authors.

Supplementary Exercises

c) Define a function *EOQK* which accepts the variable quantities from the keyboard.

d) Define a function *EOQP* which prompts for entries from the keyboard.

e) The three parameters of the eoq computation can be combined into a single vector, as, for example, `.20, 10, 100`. Define a function *EOQ* which applies to such a vector, and test it for the case *EOQ* `.20,10,100`.

20.8 Apply the function *DEFINE* to the argument *DEFTEST*, and use the techniques outlined in Session 11 to analyze it and the functions that it employs.

20.9 In the following exercise, consult the manuals for elucidation of any unfamiliar terms.

a) Define the function *F*◇÷ω, and the corresponding function *G* as follows:

```
     ∇ Z←G X
[1]    Z←÷X∇
```

b) Compare the results of *G* and *F* for various arguments, and then compare their behaviour for the argument 0.

c) Enter)*SI* to verify that the function *G* is suspended in execution and that *F* is not.

d) Suspension of *F* is prevented by the statement ⎕*TRAP*←*TRAP*, which may be seen in the canonical representation of *F* by entering ⎕*CR*'*F*'.

e) Display the value of the parameter *TRAP*, then respecify it as an empty vector (by entering *TRAP*←'') and again compare the behaviour of *F* 0 and *G* 0.

NOTES

Supplementary Exercises

20.10 Review exercise 15.5 and then (using the index to locate the relevant pages) read about the general **file access control** facilities provided by the system function ⎕*STAC*. In particular, examine the table of **permission codes** associated with ⎕*STAC*.

a) Enter ∇*TO* [⎕]∇ to display the definition of the function *TO* used to enter data in files, and identify the line that sets the access state.

b) In the line that sets the access state, replace the expression in parentheses by 1 3⍴(1↑⎕*AI*),¯1 0 and verify that any **new** file established by the modified function is private.

c) Change the function name in the header of the modified function to *PTO* (for "Private to"). Since this change destroys the function named *TO*, you may wish to enter)*COPY* 12 *COURSE TO* so that both *TO* and *PTO* are available, and then save the resulting workspace.

d) Modify the function *TO* to make the left argument of ⎕*STAC* the parameter *ACC*. Then experiment with the modified function with various settings of the parameter *ACC*.

20.11 A file (such as 13 *NATIONS* used in exercise 15.1) which was not produced by the function *TO* can nevertheless be read by the function *GET*. The ability to handle such "foreign" files complicates the definition of the function *GET* and adds to the cost of using it.

a) Display the definition of the function *GET*.

b) Enter ∇*GET*[∆10 11 12 13]∇ to remove lines 10 through 13 of the definition of the function *GET*. Verify that the more efficient modified function can still be used to read files established by the function *TO*, but fails to read a foreign file such as 13 *NATIONS*.

c) Display the definition of the modified function *GET* and compare the line numbering with the display of the original function.

20.12 The function *TUTOR* permits a student to communicate with a remote tutor who uses the complementary function *MONITOR*; the actual communication is provided by a simple and important facility called **shared variables**. The two functions may be used as an introduction to the topic of shared variables as follows:

a) Read the first four pages of the section on shared variables in the IBM manual, and look at Figure 17.

b) Enter *HELP* to obtain a discussion of the use of the function *TUTOR*.

c) Display the definitions of the functions *TUTOR* and *MONITOR*, and spend some time trying to read them without attempting to execute them.

d) To experiment with the functions, sign on two terminals on account numbers *AN*1 and *AN*2 (as discussed in Exercise 15.5) and use *TUTOR* on account *AN*1 and *MONITOR* on account *AN*2.