Before I start, I would just like to ask a question of you out there. How many of you have used APL? How many have used BASIC? Now, the toughy. How many have used APL on the 3000? No one? That is because up until last week it was not available on the 3000. We announced APL on the 3000 last week and it will be available in November.

Since I assumed that most of you were not users of APL, I thought I would start out with some very basic questions. The first question that popped to my mind as a question that someone might ask is What is APL? APL is an acronym and it stands for A Programming Language. They really went to a lot of work to figure that one out. This language was developed at IBM and has been available from IBM for the last few years. It was developed around 1967, '68, and it was not actually supported by IBM until a few years ago. APL is an interactive language. It differs from languages such as FORTRAN and COBOL in that you don't have to specifically request that the system compile your programs. Most versions of APL are interpreters. But the new version of APL available as APL\3000 is actually an interpretive compiler. It compiles code as it needs to and keeps the compiled code available in the work-space. I will get into that later.

Another question that popped to my mind is Who can use APL? Most people can use APL. It is a very easy language to learn and to start programming in APL is quite simple. We have some people working..... I should have prefaced my remarks by saying I'm from Hewlett-Packard Research Laboratories in Palo Alto where the APL program was generated. I do not work directly for the General Systems Division, but the product we developed in the labs will be available on the HP-3000 system

and will be supported by General Systems. We have one fellow who is about 12 or 13 working with us at HP. He comes in every day and he sits down at the terminal and writes APL programs. He picked up APL by sitting at a terminal and playing with it. You can start at a very elementary level because of the interactive nature of APL. You can simply type 2 + 2 and you get back an answer. You don't have to first define the fact that what you are dealing with is integers and that you want an array that contains two integers and all sorts of things like this. You don't have to get into that level of detail in order to do useful programming in APL.

The third question I have here is What is APL good for? APL is a language which because of the power built into the language, can be used in a wide variety of applications, both in the scientific world and in the business world. I will have some examples of that further on. I just wanted to go into a list of some of the features of APL that make it a powerful language. (See slide 1).

One of the first features that I have already mentioned is that APL has a calculator mode of operation in that you can sit down at a terminal and type statements in an algebraic language which are executed immediately on the machine. There are powerful primitive operators in the APL system which allow you to deal not only with scalars, which is the normal data type that most other programming languages deal with, but you can deal with arrays as an entire entity. You can define an array and then you can define operations on an array which allow you to do some very useful things.

There is another feature of APL\3000 which is only available on a few other APL's in the world. We have a virtual workspace. The concept of a workspace is one that sometimes is confusing to people. So I have a little slide here that illustrates what a workspace looks like. (See slide 2). In most languages you deal with a

series of programs that interact with data that you have stored in some other part of the system. For example, you may write a COBOL program which reads and writes into files which are available on the system. You tend to use files to hold the data and programs have to be grouped together in order to provide a useful system. But, in APL we have what is known as a workspace. An APL workspace contains not only your data but the programs which you are working with. All of these are named-objects in your workspace, and the workspace itself has a name. So what the workspace is is a collection of both program and data objects, that you need to solve a particular problem. The idea of having a workspace and collecting together everything you need to solve a particular problem makes programming much easier. You don't have to rely on the file system or link editors and segmenters and a lot of other things in order to get a programming job done in APL. So this relates back to the earlier question which I posed. What is APL good for?

The concept of a workspace and the ability to group things together like this and the interactive mode in which APL runs, allows you to write and debug programs much more rapidly than you can in any other language.

I just returned from an APL conference in Ottawa, Canada, where HP announced its APL. At this conference there was a fellow from Europe who was discussing the usage of APL in Europe. What he did was to go into several small companies that were using APL in Europe and find out the kinds of applications they were using it for. He looked at the number of lines of code generated by the various companies. He came to the conclusion that the total amount of program development time spent by these companies was three times less than it would be in any other programming language. So, what is APL good for? It is especially good for cutting down the amount of time the programmer has to be involved in actually writing an application or solving a problem in APL.

Let me go on and talk about some more of the other features of the APL\3000. I have talked about the primitive operators, the virtual workspaces. We have interactive debugging in APL. What does this mean? Let's suppose you write a program and it has a variable V. And this particular variable you have forgotten to define. You know what happens in a FORTRAN, or ALGOL, or SPL program? If you reference variable V that you haven't declared the compiler gives you an error and stops and won't let you go any further. Well, in APL when you get to the point where you are trying to use the value of V, the APL interpreter realizes there is no value for V. It prints out the line in which the particular instance of V occurred, puts a pointer under the variable V, and tells you that it has a value error. It doesn't know what the value of V is. At that point, you can enter the value of V and continue the program from where you left off. If you have a syntax error, mismatched parenthesis or something like that, at the point the APL system finds this error, it stops and points to the place at which it found the error. It tells you it has found a syntax error. You can go in and change the program where the error occurred and continue from that point or some other point in that program. As you sit and debug your programs in APL, the entire power of APL is available to you in the debugging process.

We have a built-in editor in APL. APL functions and APL character arrays can be called into the editor. You can make changes and delete lines. The flavor of the built-in editor in APL\3000 is very similar to the editor that already exists on the 3000. The editor in APL knows about APL data types and it knows about APL programs, so you never have to exit the APL system in order to edit programs or data.

Another feature that we have in APL 3000 which doesn't exist in any other APL that I know of, is that we have two languages which coexist, and operate on the same data. You can write programs in either of these two languages. The two languages are APL and APLGOL. APLGOL is an ALGOL-like control structure that is placed around APL statements and I have an example of that later on. What APLGOL tends to do is make the flow of control in your program more readily visible to someone who is reading these programs later on. You can also put comments in your APL program and that doesn't hurt you, because when the APL system recognizes a comment it simply ignores the comment from then on. The comment only takes up space off line in the disk and is never actually swapped into the main core where it would take up room. The only time it is swapped in is when you are back translating an APL program in order to edit it or change it and then you would want to look at the comments and see what they were.

The last feature of the APL\3000 that differs from other APL's is that it is a dynamic compiler. By that we mean that it compiles instead of actually interpreting the code. The first time it sees a line it has never run before it compiles some code for that line and that code is kept in the workspace. When you come back to run the function again it just reexecutes the compiled code. We have seen speed-ups on the order of two or five to one, depending on the exact problem, between a compiled and a noncompiled version. The first time you run a function it is noncompiled and it will get compiled as you run it. From then on it will stay compiled providing the types and shapes of the variables involved do not change. If they do change, the system automatically knows it has to recompile code because the code generated before is no longer valid.

I would like to show you a couple of examples of the kinds of things you can do in APL. I have one slide here that illustrates the calculator mode of operation. (See slide 3). At the bottom of the slide it shows how to calculate an average in APL. This illustrates both the calculator mode of operation and the powerful functions available because we have here a variable X which has been assigned the vector of values 100, 86, 75, 95, etc. This statement sets up the variable X and allocates storage for it to be placed in without the user having to get involved. You can then calculate the average of those numbers by summing all the elements of X and dividing by the number of elements in X. Since X is stored as an APL data type which is a vector, the APL system knows about its length and knows how to add all the elenents of that vector. These operations are available within APL and you don't have to write a separate program to do them.

To give you a little more of the flavor of APL, I have an example here of a problem that actually came up in the research lab where I work. One of the fellows wanted to calculate the following integral. It is a fairly standard integral. It is the SIN(T) over T. He wanted to find the value of that integral from zero to X where X was 3.2. Now this integral does not have a closed form solution. So one way of approximating the integral is by taking a summation. You simply divide the interval up into a number of little spaces, each one of width dt, and then you evaluate the sin at each of those points and add the whole thing together. So a series approximation for the integral, taking advantage of the fact that some common terms can be cancelled, looks something like this. (See top of slide 4). As an exercise I programmed this example in APL and BASIC.

The BASIC solution for this problem is very similar to the FORTRAN solution or the ALGOL solution, for that matter. (See middle of slide 4). First you have to declare the variables and tell the system what they are going to be. (Line 10). In this case we are going to use long floating point. Otherwise, the answer would not have been accurate enough. Then we had to initialize some variables and set up a little FOR loop which says FOR I = 1 to 4. (Lines 20 through 30). Essentially, what we did was to evaluate the integral for 4 different values of the increment of summation, DX. We wanted to compare the value of the integral we got to see how small a value of DX we needed to give us the appropriate accuracy we were looking for. So you have an outer FOR loop that does the four different values of DX and then there is another FOR loop, FOR I = 1 to L, where L is the number of things you are going to sum. (Line 70). You then do a sum which is S plus the SIN term (Line 80), and store the answer when you are done with this innermost FOR loop. (Line 100). When you are all done you can do a matrix print Y (Line 120) which in BASIC prints out the entire array of the four solutions that are shown here. So this problem is about a ten or twelve line program in BASIC.

That same solution in APL is given in one line. (See bottom of slide 4). What I did was write a function which could then be called with the different values of DX. The first three lines are a request for a display of the function FUN, and then the fourth line calls the function with the various values of DX. The answers are printed out automatically.

What I am trying to show here is that the amount of programming involved to do the same thing in APL as opposed to BASIC is significantly different. I will read the APL solution from right to left because that is the way APL operates. I generated 3.2 which was the range over which we wanted to do the summation. I took DX and divided it into 3.2 and then I took the index generator in APL; (ι) it generates all the numbers from one to the maximum number that we want. Essentially I generated the integer indices that were needed and assigned them to a variable I called XSI. Then I added a -.5 so that the series of integers becomes the series of half integers .5, 1.5, 2.5, etc., that is the denominator of the expression. The numerator of this fraction that occurs as one term in the series is the same series of half integers multiplied by DX, which is the increment size. So we take the SIN of DX times this list of half integers and divide that by the list of half integers and then the whole thing is summed using the reduction operator in APL. The result is put into R. R is the result of the function. When you run a function the result is then available to be assigned into something else. Since I didn't assign it in the line where I called the function, it simply gets printed out on the console. In APL, if you don't specifically assign something to a variable, it will get printed on your console. You don't have to write a special output statement to get things to print out for you. That is the flavor of the APL solution of this problem. I might point out some other differences.

In BASIC, the greater than sign (>) is the prompt that tells you that you are ready to input another BASIC command. However, in order to get a program which you have written to run you have to enter the command RUN. (See line following line 120). In BASIC you are normally in a function definition mode. In order to actually run a

program you have to get out of that mode and tell it to RUN. In APL you are normally in an execution mode. In order to define a function you have to flip into the function editor so that you can actually define a function.

There are matrix operations available in BASIC but you have to preface the statement by MAT to signal the BASIC interpreter that a line contains a matrix operation (e.g. line 120). In APL all of the variables here can be matrices and in most cases they are. APL treats matrices in the same way it treats scalars, etc. You don't have to do anything special to get it to handle matrices.

I have another example which is a little more lengthy, but it is fairly straightforward. It shows how APL can be used in the business community. Here is an example of the kind of things that are fairly easy to do in APL. We have here some data which is revenue for several different salesmen for several different products (See slide 5). Each number in this matrix represents the revenue generated by a particular salesman for a particular product. For example, Smith sold $140 worth of shoes, $19 worth of hats, Hall sold $659 worth of pants, etc. We have another matrix which represents the expenses each salesman incurred when selling each of those products. So these two matrices can be primitive data objects in APL. You can talk about relations between the matrices.

These matrices were entered into the APL workspace as shown in Slide 6. You simply state that the revenues are going to be a 4 x 5 matrix and here are the numbers. You input the numbers and can ask for the matrix of numbers to be printed out and it is printed out in the format that you have specified. It is stored internally as a matrix with four rows and five columns. To calculate the commissions (See slide 7) you subtract the expenses from the revenues and that is the actual

revenue each salesman has brought in. Then you multiply that revenue by the appropriate constant you are using for commissions. In a single line of APL you can generate an entire matrix of commissions for each of the salesman for each of the products.

Let's calculate the profits (See slide 8). In order to get the profits you take the revenue and subtract from that the sum of expenses and the commissions. A single APL statement calculates the profit. The result is a matrix of profits by salesman and by product line. If you want the profits by product line, what you really want is the sum over the rows, and that gives you for every row in the matrix the total amount of profit for that product. If you also want the profits by salesperson, then you do the sum over the other coordinate of the matrix which is to say you sum down the columns. And, in that case you get the profits by salesperson. If you want the total profits, that is the sum over all profits. In APL this is written +/+/PROFITS which gives you the total profit for your company. By taking advantage of the array capabilities that are available in APL, you can structure your problem in such a way that it is much easier to think about and much easier to extract information out of that data base by the operators that are available in APL.

I said earlier that we have two languages which coexist within our APL system, both APL and APLGOL. I have a brief example that shows the difference between the two languages. (See slide 9). Here are two identical programs called Toss and Flip. When you call TOSS, it prints out either "HEADS" or "TAILS" based on a random number that it gets from the APL system. In APL the function would look like this. (Top of slide 9).

In line 0 we have Toss, which is a function name, and COIN is a local variable, local to that function. COIN is assigned a random number between one and two in line 1. Then line 2 determines whether COIN is equal to 1, if so you print out the string 'HEADS'. If COIN is 2 you go down to line 5 and print out the string 'TAILS'. That is the way it would look in APL and that is probably not the clearest way to structure your program.

In APLGOL, this function looks very much like the ALGOL procedure you might write to do the same thing. If coin equals one, then print 'HEADS', otherwise print 'TAILS'. Essentially, what APLGOL does is generate the appropriate branch statements that are needed to get this function to work within the APL system. As I said, the APL and the APLGOL functions are compatible. You can call one from the other and you can write your programs in whatever language you feel comfortable with. Another fact that came out of the talk on APL usage in Europe which I referred to earlier was that people tend to write large systems of APL programs as collections of small procedures of ten and forty lines each. These procedures do specific things and they all have names, etc. This feature is very helpful in debugging your programs because you don't have to bring in long, long, long source programs of many thousand of lines and try to correct an error in line 1500 of some incredibly long program. If you keep your programs short, it is much easier to debug and to find out the flow of control within your system.

Let me just recap by going back to one of the earlier slides (See slide 1 again). After recapping, I can answer any questions you might have. The features of the APL 3000 are: 1) This is the only true calculator mode language that is available on the HP-3000. 2) It has very powerful operators built into the language. For

example, there is an operator available in APL which will sort a string of numbers in ascending order. You don't have to write your own sort procedure to do that. 3) You have virtual workspaces which allow you to collect together all the programs and data which you want to use to solve a particular application. 4) Interactive debugging which helps you write programs faster and debug them more quickly. 5) The built-in editor which allows you to edit both programs and data within your APL workspace and keep it in a fashion that is compatible with APL. 6) The coexistence of APL and APLGOL in the same workspace environment, 7) The dynamic compiler which speeds the execution of the APL programs so that you don't continually pay for the interpretive overhead.

At this point I will open it up to any questions anyone has about the system.

Q: Can you call FORTRAN subroutines from APL?

A. You cannot at this time write a FORTRAN subroutine and call it from APL. However, you can access files that may have been generated by a FORTRAN program.

Q. Do you have an interface to IMAGE?

A. We don't have a connection to image. We do have a connection to the file system, only, at this point. However, we are looking toward connections to image and other languages as a possible future extension.

Q. It sounds like APL has reduced the distinction between code and data which is built into the 3000. Is this true?

A. We found that a lot of the newer programming languages that are becoming available and a lot of the features that are required for people in the university environment, etc., tend to go away from the concept of distinct code and data. One thing, for example, you can do in APL very easily, is write an APL program which can, from the input it receives, write another APL program and go off and execute that program.

This is one of the reasons why we went to APL. I might just add that one of the reasons why I got interested in APL originally is because I had an application where, based on information I got from the user, I needed to write a program which would then do some special things for the user. I wrote an application in about three months in APL which then took about eighteen man months to translate into ALGOL. What this program does is take input from the user and generates an ALGOL program which runs on our system. That was a much more difficult task in ALGOL than in APL. However, there is still a distinction between code and data in your workspace. It is a fairly rigid distinction. You can turn a piece of code into a piece of data and then work on it as a piece data and then turn it back into a piece of code. There are safeguards in the APL system that do not let you read the code directly as data; you have to actually go through a system operator that will turn it from a piece of code into a piece of data. It will still give you protection.

Q: How do you interact with the file system in APL?

A: The way we interact with the file system is through a concept known as shared variables. You can take a variable in your workspace and specify that this variable is shared with a particular file on the 3000. Then every time you assign to this variable you put something out to the file and everytime you read from the variable you read something in from the file. Essentially, we give you access to all of the file controls which are available on the 3000.

The reason we have not put a lot of effort into designing a file system, as many of the other APL systems have, is because we felt that since we give you a virtual workspace, you should be able to store all the data and programs you need in

the workspace without having to go to a file to do it. We also wanted to give you the full capability of the MPE file system. We don't have any special operators which store an object as an APL data object which then cannot be read by anyone else unless they know what APL data objects look like.

Q: Is APL good for writing simulators?

A: APL is very well suited for writing simulators. As a matter of fact, one of the things that one of our fellows in the lab did for fun was to write a simulator for the INTEL 8080 microprocessor. The simulator has all the commands of the microprocessor as an APL function. He can run that simulator in APL and he can also produce 8080 machine instruction. The HP-2641 terminal has an 8080 in it and you can actually download instructions into the 8080 and run them right there. This was kind of an interesting APL program which wrote programs for the terminal and could be used to change the operations of the terminal. We are not necessarily recommending that you do that.

Q: What kind of protection is available in APL?

A: When you load your workspace, you load in all your functions. If you want to copy a particular function, you can say)COPY followed by a list of functions from the particular workspace. APL will tell you which ones it couldn't copy. There is a protected copy which will not wipe out a function by the same name in your workspace and if you use that feature APL will tell you which objects it was unable to copy.

If you lock a function, then no one can get in and alter that function.

We have some interesting additions to our APL system in the area of local variables. If you have a variable declared as local in a function, it will shadow whatever value the global variable had when you are running inside the function.

However, it is possible during debugging or under program control to go back and find out the value of the global variable. In most APL systems that is not possible, but in ours it is.

Q: How much do you pay for all the power of APL?

A: You do pay some costs for the interpretive overhead, i.e. the ability APL gives you to fix up your programs when you have an error. How much you pay depends on your particular application. Again, let me refer to the paper on the European APL study. They found in some systems the costs was essentially the same, because the amount they spent on additional CPU time, they saved on being able to program their solutions more quickly. A typical program only took about three or four minutes of CPU time to run, but it had to be run once a month and updated continuously. They found that APL was cost effective in that case. In general, you do pay something in overhead for all the power available to you, but it is not that bad.

Q: How big is APL?

A: APL is a subsystem on the 3000 similar to BASIC, etc. It takes a fair amount of code, but code is shared between all APL users.

Let me just say I will be giving a demonstration of the APL 3000 on our 2641 APL terminal at one of the birds of the feather sessions this evening. If you have further questions please come to that session and we will give you some online demonstrations of the system over the phone lines to Palo Alto. I will also be glad to answer your questions after this session. Right now we are about out of time.

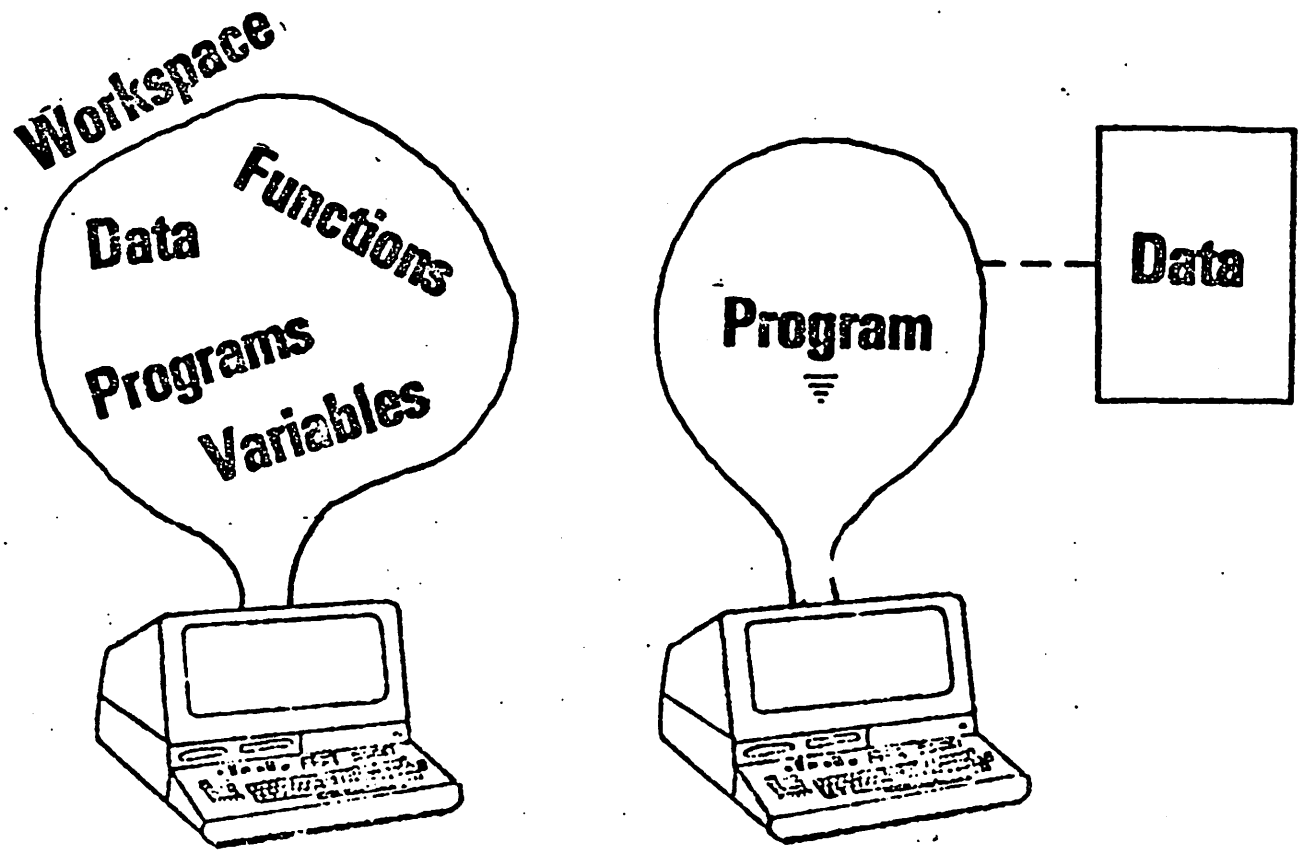
Features of APL 3000

1. Calculator Mode Operation
2. Powerful Primitive Operators
3. Virtual Workspaces
4. Interactive Debugging
5. Built-in Editor
6. APL and APLGOL
7. Dynamic Compiler

SLIDE 1

HEWLETT PACKARD

APL



HEWLETT hp PACKARD

APL

* MANY OPERATORS AND FUNCTIONS
* NO HIERARCHY OF OPERATION
* HANDLES ARRAYS AS EASILY AS SINGLE NUMBERS
* CALCULATOR MODE OPERATION

CALCULATING AN
AVERAGE IN APL

```
      X←100 86 75 95 73 97 85 90
      AVERAGE←(+/X)÷⍴X
      AVERAGE
87.625
```

SLIDE 3

$$Si(3.2) \simeq \sum_{N=1}^{(N-\frac{1}{2})dx \leq 3.2} \frac{SIN[(N-\frac{1}{2})\,dx]}{(N-\frac{1}{2})}$$

Basic Solution

```
>LIST
ARPADB
  10 LONG D,S,Y[4]
  20 D[1]=.1
  21 D[2]=.01
  22 D[3]=.001
  23 D[4]=.0001
  30 FOR I=1 TO 4
  50   L=CEI(3.2/D[I])
  60   S=0
  70   FOR N=1 TO L
  80     S=S+SIN((N-.5)*D[I])/(N-.5)
  90   NEXT N
 100   Y[I]=S
 110 NEXT I
 120 MAT PRINT Y
>RUN
ARPADB
 1.851523548519127L+00          1.851402182774109L+00
 1.851400917321872L+00          1.851400899101099L+00
```

APL Solution

```
      ⎕CR'FUN'
 R←FUN DX
 R←+/(1○DX×XSI-.5)÷⁻.5+XSI←⍳⌈3.2÷DX

      FUN .1 ◇ FUN .01 ◇ FUN .001 ◇ FUN .0001
1.851528522
1.851402173
1.85140091
1.851400897
```

SLIDE 4

HEWLETT PACKARD

APL

EXAMPLE

AS A SALES MANAGER OF A SMALL COMPANY
YOUR REVENUES AND EXPENSES ARE:

REVENUES

	JONES	*SMITH*	*WALL*	*HARRI*	*HALL*
SHOES	190.00	140.00	1926.00	14.00	143.00
HATS	325.00	19.00	293.00	1491.00	162.00
PANTS	682.00	14.00	852.00	56.00	659.00
BOOTS	829.00	140.00	609.00	120.00	87.00

EXPENSES

	JONES	*SMITH*	*WALL*	*HARRI*	*HALL*
SHOES	120.00	65.00	890.00	54.00	430.00
HATS	300.00	10.00	23.00	802.00	235.00
PANTS	50.00	299.00	1290.00	12.00	145.00
BOOTS	67.00	254.00	89.00	129.00	76.00

SLIDE 5

HEWLETT hp PACKARD

APL

THESE WERE ENTERED INTO AN APL WORKSPACE AS FOLLOWS:

```
      REVENUES←4 5⍴190 140 1926 14 143 325 19 293 1491 162.□
□:
      682 14 852 56 659 829 140 609 120 87

      REVENUES
  190  140  1926    14  143
  325   19   293  1491  162
  682   14   852    56  659
  829  140   609   120   87
```

SLIDE 6

APL

```
      EXPENSES
120    65   890    54   430
300    10    23   802   235
 50   299  1290    12   145
 67   254    89   129    76

      REVENUES
190   140  1926    14   143
325    19   293  1491   162
682    14   852    56   659
829   140   609   120    87
```

COMMISSION

```
      .062×0⌈REVENUES-EXPENSES
 4.34    4.65   64.232    0        0
 1.55     .558  16.74    42.718    0
39.184   0       0        2.728   31.868
47.244   0      32.24     0         .682
```

```
COMMISSION←.062×0⌈REVENUES-EXPENSES
```

HEWLETT hp PACKARD

APL

PROFITS

```
      PROFITS←REVENUES-EXPENSES+COMMISSION

      PROFITS
 65.66     70.35    971.768   ¯40      ¯287
 23.45      8.442   ¯253.26   646.282   ¯73
592.816   ¯285      ¯438       41.272   482.132
714.756   ¯114       487.76    ¯9       10.318
```

PROFITS BY PRODUCT LINE

```
      +/PROFITS
780.778  858.434  393.22  1089.834
```

PROFITS BY SALES PERSON

```
      +/PROFITS
1396.682  ¯320.208  1274.788  638.554  132.45
```

TOTAL PROFITS

```
      +/+/PROFITS
3122.266
```

SLIDE 8

APL\3000

TWO IDENTICAL PROGRAMS
THE APLGOL ONE IS EASIER TO READ

APL

```
[0]     TOSS;COIN
[1]     COIN←1?2
[2]     →(~COIN=1)/T
[3]     'HEADS'
[4]     →0
[5]    T:'TAILS'
```

```
      TOSS
TAILS
      TOSS
HEADS
      TOSS
HEADS
      TOSS
TAILS
      TOSS
TAILS
```

APLGOL

```
[0]    PROCEDURE FLIP,COIN;
[1]      COIN←1?2;
[2]      IF COIN=1 THEN
[3]        'HEADS'
[4]      ELSE
[5]        'TAILS';
[6]    END PROCEDURE
```

```
      FLIP
HEADS
      FLIP
TAILS
      FLIP
HEADS
      FLIP
TAILS
```