Unit 6
Arithmetic

Unit Objectives

After completing this unit, you should be able to:

- Use the **`expr`** utility
- Understand **`expr`** arithmetic and logical operators
- Use shell **`let`** or **`(( ))`**
- Use number bases
- Use **`let`** logical operators
- Use integer variables
- Use implicit **`let`**
- Understand the **`bc`** utility

expr Arithmetic

AIX provides the **expr** utility to perform ***integer*** arithmetic

expr argument1 operator argument2 …

expr features

- Runs as an external executable
- Writes results to standard output
- Exit code is 0 for non-zero evaluations
- Exit code is 1 for zero or null evaluations
- Exit code is ≥ 2 if an expression is invalid
- Mostly used for control flow in shell scripts – loop counters

expr Arithmetic Operators

To group expressions use:

( )	fixes evaluation order - otherwise normal rules of precedence apply

The integer operators result in mathematical evaluations:

=	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

expr Examples

Here is some simple integer arithmetic...

```
$ var1=6; var2=3
$ expr  $var1  /  $var2
2
$ expr  $var1  -  $var2
3
$ expr  \(  $var1  +  $var2  \)  \*  5
45
$ _
```

What is the result of the following?

```
$ expr  10  %  3

$ expr  10  /  3
```

The let Command

```
let argument ..

-or-

((  argument  ))
```

- The **let** built-in shell command performs long integer arithmetic approximately 10 times faster than **expr**
- Evaluates each argument as an arithmetic expression
- No quotes for special characters, or arguments with **spaces** or **tabs** in them, within **((...))**
- Variables need no **$**
- The exit code is 0 (true) for non-zero, and 1 (false) for zero evaluations
- In ksh93, **let** will use decimal numbers, if you give the arguments in decimal notation
- In bash and ksh88, integer only

let Arithmetic Operators

For simple arithmetic:

`( )`	overrides normal precedence rules
`*`	multiplication
`/`	division
`%`	remainder
`+`	addition
`-`	subtraction (or unary minus)
`=`	assignment

`var op= exp` means `var = var op exp`

Up to nine levels of nested processing will be evaluated:

```
$ z=2  ;  y="z + 1"
$ ((  x=3*y  ))
$ print  $x
9
$ _
```

let Arithmetic Examples

Some simple arithmetic...

```
$ a=1 b=2
$ ((  z = 2#10 + -b ))
```

unary minus needs a space before it, not after

```
$ let  c=a+b  d=b\*b
```

no spaces, but \ needed for * multiple arguments

```
$ (( e  =  9 / b ))
```

integer division

```
$ (( e  +=  a ))
```

assignment: addition

```
$ print $z $a $b $c $d $e
```

What do you think we get?

What is the difference between these?

`$(( ... ))` and `(( ... ))`?

let Logical Operator

Logical expressions evaluate to 1 if true, 0 if false
(the exit code is 0 for non-zero, 1 for zero – as expected):

!	logical negation
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to
&&	logical "and" = 1 if both LHS and RHS are true (RHS not evaluated if LHS is false)
||	logical "or" = 1 if either LHS or RHS are true (if LHS is true, RHS not used)

let Logical Examples

```
$ (( p = 9 ))

$ (( p = p * 6 ))
$ print $p
54

$ (( p > 0 && p <= 10 ))
$ print $?
1

$ q=100
$ (( p < q || p == 5 ))
$ print $?
0

$ if (( p < q && p == 54 ))
> then
> print TRUE
> fi
TRUE

$ _
```

base#number Syntax

With **let** you are not limited to just decimal (base ten) integers:

- **let** constants are of the form **base#number**
- **base** is an integer in the range 2 to 36 (10 default)
- **number** may include upper or lowercase letters for bases greater than 10

2#100 in binary	=	4 (in base 10)
8#33 in octal	=	27
16#b in hexadecimal	=	11
16#2A in base16	=	42

Shell integer Variables

Shell variables are stored as character strings unless defined with the **integer** command

```
integer variable=value ...
```

-or-

```
typeset -iN variable=value ...
```

- Sets the **integer** attribute for each variable
- **typeset** can define a base **N**, variables then *print* in the specified base (2 to 36)
- Assignment to an **integer** variable causes expression evaluation – an implicit **let** command
- **let** does not have to convert **integer** variables from character strings to numerical values

integer Examples

Some examples of integer and typeset -i ...

```
$ integer x                          x can hold only integers
$ x=string
ksh:  string: 0403-009 The specified number is
not valid for this command.
$ x=5+10                             implicit let command
$ print $x
15
$ (( x = 5 + 100 ))
$ print $x
105
$ typeset -i8 nums0 nums1 nums2
$ nums0=8#5                          define an octal integer variable
$ nums1=8#10
$ (( nums2=8#3*nums0 ))              assign value
$ print ${nums2}
8#17
$ x=${nums2}
$ print $x                           print gives answer in base 10
15
$ _
```

Implicit let Command

integer variable assignments are an implicit **let** command
Other implicit **let** commands are:

- Values for the **shift** command

```
shift OPTIND-1
```

- Resource limits with **ulimit**

```
ulimit -t TMOUT+60
```

bc - Mathematics

The AIX system provides the **bc** utility

```
bc  [file]
```

- Performs floating point arithmetic
- Acts as a filter command or interactively
- Reads arithmetic expression strings from standard input or from a specified file
- Semicolons or new lines separate expressions
- Sets the **scale** variable inside **bc** to define the required number of decimal places
- Prints results to standard output

bc Operators

For simple arithmetic and logical evaluations, use:

`(, ), +, -, *, /, %, =`	as for **`let`** arithmetic operators
`==, !=, <, <=, >, >=`	as for **`let`** logical operators
`x^y`	raise **`x`** to the power **`y`**
`sqrt(x)`	square root
`x++ ++x`	post and pre increment **`x`**
`x-- --x`	post and pre decrement **`x`**
`x op= y  ≡  x = x op y`	for +=, -=, *=, /=, %=, ^=

A library provides complex mathematical functions:

`s(x)`	sine of **`x`**
`c(x)`	cosine of **`x`**
`e(x)`	natural exponential of **`x`**
`l(x)`	natural log of **`x`**
`a(x)`	arctangent of **`x`**
`j(n,x)`	Bessel function

Precision functions:

`length(n)` number of significant digits for example, 123.456 has n=6

`scale(n)` number of digits after decimal point for example, 123.456 has n=3

bc Examples

Here are some examples of **bc** working both as a filter and interactively.

```
$ print '1/4' | bc
0
$ print 'scale = 3 ; 1/4' | bc
0.250
$ print '5.5 * 2.2' | bc
12.1
$ value=$( '5.5 * 2.2' | bc )
$ print $value
12.1
$ bc

sqrt(4)
2
Ctrl-d
$ _
```

integer division without a scale

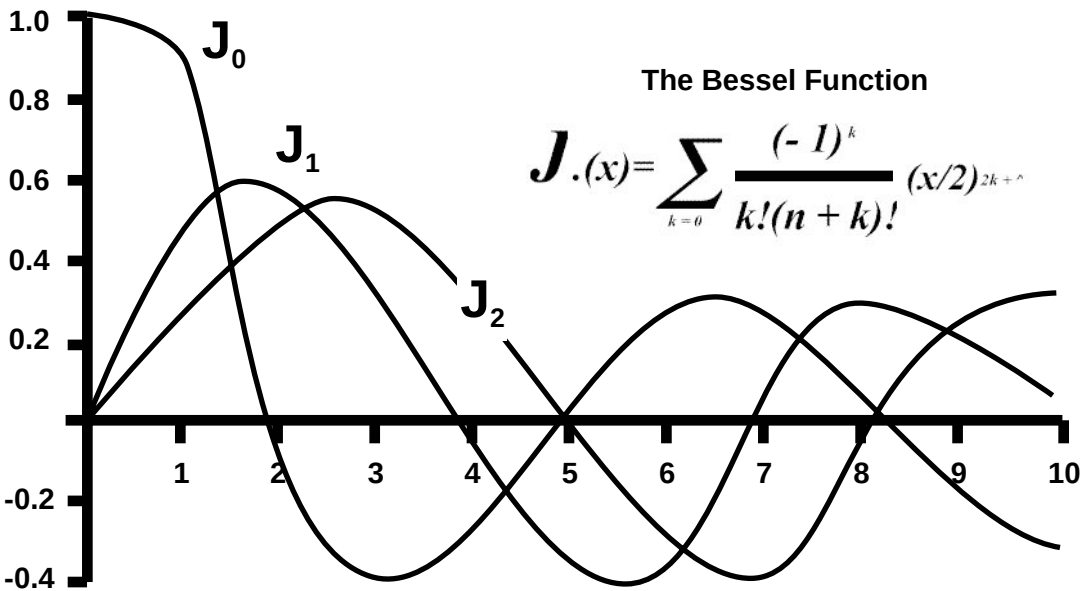
explicit scale value set

scale set implicitly from input

assign the answer to a variable

no prompt – this is my input
the result from the command
to end interactive mode

An Example of the Power



The Bessel Function

$$J_n(x) = \sum_{k=0} \frac{(-1)^k}{k!(n+k)!} (x/2)^{2k+n}$$

Checkpoint

1. Multiply together variables **a** and **b**, using **expr**.
2. Use **expr** to multiply variable **a** by the sum of **b** and **c**.
3. Set variable **hex** to contain the hexadecimal value **7c**.
4. Write a **let** statement to test whether variable **a** is smaller than variable **b**.
5. Define a variable **num** as numeric only.
6. Increment a numeric variable **numvar**, by three.
7. How would you calculate 6/7 to 6 decimal places?
8. How would you calculate the square root of 8541976320?

Unit Summary

- The **`expr`** utility
- **`expr`** arithmetic and logical operators
- Shell **`let`** or **`(( ))`**
- Number bases
- **`let`** logical operators
- Integer variables
- Implicit **`let`**
- The **`bc`** utility