Advanced Unix Tools	**Lecture 6**
CS214 Spring 2004	**Friday March 5, 2004**

Here redirection

Recall that redirection allows you to redirect the input to a command from a file (using $<$), or the output to a file (using $>$, $>>$, depending on whether we want to overwrite the file or append to it). A special form of redirection is useful, called a "here document". It allows you to redirect input not from a file but from explictly provided data. The general form of this form of redirection is as follows:

```
cmd << FLAG
line1
line2
line3
line4
FLAG
```

This executes *cmd*, feeding it *line1*, *line2*, *line3* and *line4* as input. The *FLAG* after $<<$ indicates until where to read the input; the first line containing *FLAG* by itself is the end of the input to feed to the command. The terminating expression need not be *FLAG*. It can be anything. Generally, it will be something that does not appear in the text to feed to the command. Note that expansion is performed on the lines to pass to the command. Expansion is *not* performed on *FLAG*.

Here's a concrete example, to mail a piece of text to *joe*:

```
mail -s "value of path var" joe << XYZ
Joe, here's the value of my
PATH variable: $PATH
XYZ
```

Case statement

The `case` statement is sometimes a nice alternative to conditionals, as it allows for some amount of pattern matching.

```
case word in
  patt | ... | patt)
    cmd
    cmd
    ;;
  patt | ... | patt)
    cmd
    cmd
    ;;
esac
```

As usual, the argument *word* is expanded according to the shell rules. Then, the system goes through each clause and sees if *word* matches one of the patterns in the clause. (Patterns in a clause are separated by |.) A pattern uses the same special symbols as pathname expansion (`*`, `?`, `[...]`, ...). For example, both `unix` and `linux` match the pattern `*n?x`. The first time a match is found, the corresponding commands are executed. A typical example of use of `case` is to conditionally set variables depending, say, on the operating system you are running. To get a hold of the operating system, you can either refer to the `bash` environment variable `OSTYPE`, or use the unix command `uname`. Let's use the latter. The command `uname` returns information about the system you are currently running. Command-line options controls the exact information returned. What we want is `uname -s`, which returns the name of the operating system:

```
case ‘‘$(uname -s)’’ in
in
  ?inux)
    echo "You’re running linux..."
    opsystem=’’LINUX’’
    ;;
  Sun*)
    echo "You’re running SunOS..."
    opsystem=’’SUNOS’’
    ;;
  *)
    echo "Unrecognized operating system..."
    exit 1
    ;;
esac
```

Notice the use of the pattern `*` as a catch-all default case.

Arithmetic expressions

The `bash` shell allows some amount of arithmetic expressions to be evaluated directly. Two forms are used. The first is a substitution form. The shell will expand the form `$((` *expr* `))` by evaluating the arithmetic expression *expr* and substituting the result. Arithmetic is performed using long integers, without checks for overflow. Division by zero is reported as an error. Note that *expr* is treated as though it were inside double-quotes; hence, variable and command substitution can occur, but not pathname expansion.

Operations are reminiscent of C operations. Logical values for integers is as in C: true is zero, false is non-zero. These operations are listed in the decreasing order of precedence:

`- +`	unary minus, plus
`! ~`	logical, bitwise negation
`**`	exponentiation
`* / %`	multiplication, division, remainder
`+ -`	addition, subtraction
`<< >>`	bitwise shifts
`<= >= < >`	comparisons
`== !=`	equality, inequality
`&`	bitwise AND
`^`	bitwise exclusive OR
`|`	bitwise OR
`&&`	logical AND
`||`	logical OR
`expr?expr:expr`	conditional evaluation
`= *= /= %= += -=`	assignment

Parentheses can be used to affect the order of operations, as usual. If you use an identifier such as `foo` in the arithmetic expression, the value of the corresponding shell variable is used. The value is coerced into an integer. If the value does not correspond to an integer, 0 is used.

Consider the following simple examples:

```
$ echo $((1 + 1))
2
$ foo=10
$ echo $((foo))
10
$ echo $((foo * 2))
20
$ echo $((foo = 40))
```

```
40
$ echo $foo
40
```

Notice the use of assignment in `echo $((foo = 40))`. The variable `foo` is assigned value 40, which has the side-effect of changing the value of the shell variable `foo` to 40. Hence, arithmetic evaluation can affect the value of shell variables, which can be quite useful.

Note that variable substitution occurs in `$((...))` before arithmetic evaluation (refer to lecture 2). A priori, there is no difference between `$((foo + 20))` and `$(($foo + 20))`, except that in the latter case, the shell performs variable substitution before performing the arithmetic evalation, so that if `foo` has value 10, then the shell will in fact evaluate `$((10 + 20))`, never seeing that 10 came from a shell variable. This can lead to some pretty interesting effects. If you write `$(($foo = 20))`, you can put the name of a variable to update in `foo`, and variable substitution will plug it in the arithmetic expression before evaluating it. So you can parametrize an expression by the variables it updates. For instance:

```
$ a=10
$ b=20
$ c=30
$ foo=a
$ echo $(($foo=5))
5
$ echo $a
5
$ echo $b
20
$ echo $c
30
```

An alternative form exists, typically used to assign the result of evaluating expressions to variables. The shell command `let` simply evaluates its arguments as arithmetic expressions. Since no substitution occurs, the only way for this to be useful is for its side-effects, i.e., the assignment of values to variables. For example:

```
$ a=10
$ let 'c=a*2+a*3'
$ echo $c
50
```

If you have more complicated arithmetic to perform, say, with floating point numbers, then you can revert to an external command. The command `bc` implements a fairly complete calculator, evaluating expressions from `stdin` and returning the results to `stdout`. (Although some amount of care is needed to catch errors, etc.) Invoking `bc -l` will provide access to the full standard math library. For instance, to compute the sine of 3.4 and assigning it to shell variable `result`, you can use:

```
$ result=$(echo 's (3.4)' | bc -l)
$ echo $result
-.25554110202683131924
```