Shells

• A shell is a command line interpreter that is the interface between the user and the OS.

• The shell:
  – analyzes each command
  – determines what actions are to be performed
  – performs the actions

• Example:

  \texttt{wc \textasciitilde l \texttt{file1} > \texttt{file2}}
Which shell?

- **sh** – Bourne shell
  - Most common, other shells are a superset
  - Good for programming
- **csh or tcsh** – command-line default on CDF
  - C-like syntax
  - Best for interactive use. Not good for programming.
- **bash** – default on Linux (Bourne again shell)
  - Based on sh, with some csh features.
- **korn** – written by David Korn
  - Based on sh – Some claim best for programming.
  - Commercial product.
bash versus sh

• On the CDF machines, when you run `sh`, you are actually running `bash`.
• `bash` is a superset of `sh`.
• For CSC209, you will be learning only the features of the language that belong to `sh`. 
Common shell facilities

• Input-output redirection
  
  prog < infile > outfile
  ls >& outfile      # csh stdout and stderr
  ls > outfile 2>&1  # sh stdout and stderr

• Pipelining commands
  – send the output from one command to the input of the next.
  
  ls -l | wc
  ps -aux | grep krueger | sort
Job Control

- A job is a program whose execution has been initiated by the user.
- At any moment, a job can be running or suspended.
- Foreground job:
  - a program which has control of the terminal
- Background job:
  - runs concurrently with the parent shell and does not take control of the keyboard.
- Start a job in the background by appending &
- Commands: ^Z, jobs, fg, bg, kill
File Name Expansion

ls *.c
rm file[1-6].?
cd ~/bin
ls ~krueger
ls *.[^oa]  - ^ in csh, ! in sh

- * stands in for 0 or more characters
- ? stands in for exactly one character
- [1-6] stands in for one of 1, 2, 3, 4, 5, 6
- [^oa] stands in for any char except o or a
- ~/ stands in for your home directory
- ~krueger stands in for krueger’s home directory
Exceptions

• `ls .*` doesn’t do what you would expect

Why?
  – `.*` matches . and ..
  – because . files are hidden files, we don’t usually want to include them in our operations.

• How to get around this feature?
  – `ls -d .*` - still catches . and ..
  – `ls .??*` - misses files like .b

Challenge: find other ways.
Shell Programming
(Bourne shell)

- Commands run from a file in a subshell
- A great way to automate a repeated sequence of commands.
- **File starts with** `#!/bin/sh`
  - absolute path to the shell program
  - not the same on every machine.
- Can also write programs interactively by starting a new shell at the command line.
  - Tip: this is a good way to test your shell programs
Example

• In a file:
  ```
  #! /bin/sh
  echo "Hello World!"
  ```

• At the command line:
  ```
  skywolf% sh
  sh-2.05b$ echo "Hello World"
  Hello World
  sh-2.05b$ exit
  exit
  skywolf%
  ```
Commands

- You can run any program in a shell by calling it as you would on the command line.
- When you run a program like `grep` or `ls` in a shell program, a new process is created.
- There are also some built-in commands where no new process is created.

- `echo`
- `set`
- `read`
- `exit`
- `test`
- `shift`
- `wait`

"man sh" to see all builtins.
Variables

• local variables – spaces matter
  – name=value – assignment
  – $name – replaced by value of name
  – variables can have a single value or list of values.

• Single value:
  bindir="/usr/bin"

• List of values (separated by spaces):
  searchdirs="~/tests $HOME/test2 ."
Example:
($ is the default sh prompt)

$ bindir="/usr/bin"
$ searchdirs="~/tests $HOME/test2 ."
$ echo $searchdirs
~/tests /u/krueger/test2 .
$ echo $bindir
/usr/bin
String Replacement

• Scripting languages are all about replacing text or strings, unlike other languages such as C or Java which are all about data structures.

• Variables are placeholders where we will substitute the value of the variable.

• Example:

```bash
iters="1 2 3 4"
for i in $iters; do ==
    echo $i
done

for i in 1 2 3 4; do
echo $i
done
```
Quoting

• Double quotes inhibit wildcard replacement only.

• Single quotes inhibit wildcard replacement, variable substitution and command substitution.

• Back quotes cause command substitution.

• Practice and pay attention.
Quoting example

$ echo Today is date
Today is date

$ echo Today is `date`
Today is Thu Sep 19 12:28:55 EST 2002

$ echo "Today is `date`"
Today is Thu Sep 19 12:28:55 EST 2002

$ echo 'Today is `date`'
Today is `date`
Another Quoting Example

• What do the following statements produce if the current directory contains the following non-executable files?

```
$ echo *
$ echo ls *
$ echo `ls *`
$ echo "ls *
$ echo 'ls *
$ echo `*`
```

- double quotes
- single quote
- back quote
More on Quoting

- Command substitution causes another process to be created.
- Which is better? What is the difference?

```
src=`ls *.c`
```

or

```
src="*.c"
```