Shell Performance, Convenience, and Compatibility

Introduction & Intent

The intent of this paper is to provide a cross between a white paper and a QuickSheet - A paper that provides some quick reference, but with some context and more description than a cheat-sheet format. (Note: A quicksheet on the subject has been created.)

When writing scripts, the shell programmer frequently needs to make trade offs between shell performance and compatibility. For the simple script this is not much of an issue, but when coding loops that can iterate a large number of times or large script frameworks that can call upon many sub-scripts the performance penalty can be noticeable.

There are modern shell language conventions that save the cost of executing a sub-process that were once required of older shells. The task of creating a sub-string, finding a pattern match, or math functions no longer require creating a entirely new process structure.

Unfortunately, many Unix administrators have been conditioned to write to the lowest common denominator to insure portability of either their scripts or their skills. Even if the script someone is working on will not run across multiple systems, the library of working options has been whittled down to fit on as many implementations as possible.

This paper is not meant to be an exhaustive discussion on shell standards or a hardcore analysis of compute cycles of one method over another. Those technical discussions are beyond the needs of most shell programmers. One truth of shell programming is that a small subset of the community picks nits over precise appeals to standards, while the far greater majority does not dwell on these issues. I have no intentions of wading into the level of discussion maintained by the standards people, I am trying to speak to the rest of us who wonder why our script broke, acted in an unpredictable way, or just ran like crap.

So this paper offers a few shell conventions of convenience / performance, where they might be used, and where they should not.

Shells tested

I have only considered Bourne and Bourne derived/compatible descendants. Furthermore, because of a more big-Unix approach, this paper is more Korn-centric and less Bash oriented. Bash was evaluated on the systems that include it natively. C(sh), or some of the more exotic shells are not considered (primarily because of the clear syntactical differences).

The choice of operating environments were those that are most common Unix environments.

OS-X	Tests were run on 10.6.3 (Snow Leopard). There are three Bourne-ish variants on OS-X: Bourne, Korn, and Bash. The Korn is a 93 version (Version M 1993-12-28 s+). All three are distinct implementations, and not simply linked to each other. OS-X was chosen as it is where most development efforts I work on are managed from. I don't consider it a datacenter OS, but many Unix admins may work from (develop on) this platform.
AIX	I tested this primarily on a 5.3 system, but then validated on 6.1 TL4 - The results were the same. AIX hard links sh and ksh, and the binary does not appear to act differently based on how it is called. The sh/ksh version is Korn 88 based. IBM ships a Korn 93 called /bin/ksh93. There is no Bash native to (BOS-installed on) AIX. AIX does ship an actual Bourne shell as /usr/bin/bsh. bsh was not tested in this paper - primarily because AIX *represents* the Korn shell as Bourne through the linking of sh to ksh. AIX 7.1 offers an

updated Korn93 shell that has been tested, but is not included here until the official release of 7.1. (The good news is that the 7.1 Korn 93 is *very* up to date, and passes virtually every test.)

Solaris	Tests were run on Sparc Solaris 10. All copies of sh on Solaris are soft links to /sbin/sh. The default ksh is Korn 88, while /usr/dt/bin/dtksh is (used as) the Korn 93 implementation. The standards compliant (/usr/xpg4/bin/)sh shell was not tested.
Linux	I used CentOS 5.4 for these tests with a fairly typical install. /bin/ksh is a (series of) soft link(s) to /bin/ksh93. /bin/sh is a soft link to /bin/bash.
FreeBSD	I tested against FreeBSD 8 and used the latest ports (as of May 2010). The shells were Bourne, Korn 88 - (pd)ksh, Korn 93 - ksh93, and Bash. The results tended to be varied as each shell handled the tests differently because the shells were four actual implementations rather than links to a lesser set.
HP-UX	The tests only included the two default shells Bourne and Korn 88 that shipped with 11i v3. The two shells agreed on every test, yet they are two distinct binaries. The difference between the two was not uncovered using this limited set of tests. Finally, /bin is a link to /usr/bin, so the shells "exist" in both places.

Concatenate Operator

Description: Combine strings using assignment operator syntax

Example:

```
var=
var+=c
var+=a
var+=t

echo var ${var}
```

Expected Results:

```
var cat
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	√	X	X	√	X	X
ksh	√	X	X	√	X	X
ksh93	∅	X	X	√	√	∅
bash	√	∅	X	√	√	∅

Notes: The concatenate operation is technically a ksh93 convention. It is designed to concatenate string variables or increment numeric values.

Concatenate Operator With Array

Description: Use the concatenate operator to add individual items to an array

Example:

```
myarray=( P A )

myarray+=( S )
myarray+=( S )

echo ${myarray[*]}
```

Expected Results:

```
P A S S
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	√	X	X	√	X	X
ksh	√	X	X	√	X	X
ksh93	∅	X	X	√	√	∅
bash	√	∅	X	√	√	∅

Notes: The *portable* alternative is to assign the array and addition to itself. For example:
`myarray=( "${myarray[@]}" "new" "items" )`
Note that the `"${myarray[@]}"` syntax is used to preserve any whitespace in the array members that `${myarray[*]}` would interpret as different array items.

Numeric Base Conversions

Description: Convert the base of a number using the shell (as opposed to something like bc)

Example:

```
hex=ff
typeset -i 10 decA=16#${hex}

oct=377
typeset -i 10 decB=8#${oct}

echo ${decA} ${decB}
```

Expected Results:

```
255 255
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	X	√	X	X	X	√
ksh	√	√	√	√	√	√
ksh93	∅	√	√	√	√	∅
bash	X	∅	X	X	X	∅

Notes: The test code performs two conversions using two different hex values.

Language-based Test Convention

Description: The double bracket test is a language convention instead of a builtin or command

Example:

```
var=

if [[ -z ${var} ]]
then
    echo var is null
fi
```

Expected Results:

```
var is null
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	√	√	X	√	X	√
ksh	√	√	√	√	√	√
ksh93	∅	√	√	√	√	∅
bash	√	∅	√	√	√	∅

Notes: Because the [[is part of the language, and not a (builtin or external) *command*, the way in which contents are treated is different. For example, variables interpreted by the shell can be understood as variables while variables interpreted by a command must be parsed as strings. This is why potentially empty variables do not need to be quoted - as in the above example.

Substring Pattern Matching

Description: Access substrings using language conventions rather than externals such as sed or cut

Example:

```
var=beginXend
echo ${var#begin}
echo ${var%end}
```

Expected Results:

```
Xend
beginX
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	√	√	X	√	√	√
ksh	√	√	√	√	√	√
ksh93	∅	√	√	√	√	∅
bash	√	∅	√	√	√	∅

Notes: The example here is the most simplistic implementation that chops patterns off the beginning and end.

Basic Conditional Pattern Matching

Description: This is a series of simple pattern matches used in conditionals that will search for a NEEDLE in a HAY-string

Example:

```
STRING=HAYNEEDLEHAY

if [[ ${STRING} = *NEEDLE* ]]
then
  echo PASS
fi

if [[ ${STRING} =
HAY@(NEEDLE|NOODLE)HAY ]]
then
  echo PASS
fi

if [[ ${STRING} != *NOODLE* ]]
then
  echo PASS
fi

if [[ ${STRING} = *(OATS)HAY* ]]
then
  echo PASS
fi

if [[ ${STRING} = HAY?(-)NEEDLE?
(-)HAY ]]
then
  echo PASS
fi

if [[ ${STRING} = +(HAY)*+(HAY)
]]
then
  echo PASS
fi
```

Expected Results:

```
PASS
PASS
PASS
PASS
PASS
PASS
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	X	√	X	X	X	√
ksh	√	√	√	√	√	√
ksh93	∅	√	√	√	√	∅
bash	X	∅	X	X	X	∅

Notes: The tests are as follows:

- Look for NEEDLE in the string
- Look for either HAYNEEDLEHAY or HAYNOODLEHAY
- Look for a string that does not contain "NOODLE"
- Look for a string that contains HAY that is possibly preceded by one or more OATS

- Look for the HAYNEEDLEHAY string that may be partially/all hyphenated
- Look for a string that begins and ends with at least one HAY

It should be noted that not all of these tests failed on all platforms. A success of failure listed here means that all tests either passed or failed.

Also note that I used single = instead of double == in these (and other) examples. The double == is the preferred Korn 93 convention. This set of tests was run both ways without any changes to the results.

Repeat Sequence Pattern Matching

Description: This syntax matches min / max instances of a pattern

Example:

```
STRING=HAYNEEDLEHAY

if [[ ${STRING} = *{2}(E)* ]]
then
  echo PASS
fi

if [[ ${STRING} != *{3}(E)*
]]
then
  echo PASS
fi
```

Expected Results:

```
PASS
PASS
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	X	X	X	X	X	X
ksh	√	X	X	√	X	X
ksh93	∅	X	X	√	√	∅
bash	X	∅	X	X	X	∅

Notes: The syntax allows for {m,n} and {n}. Both examples here use {n} and are looking for a specific count of the letter 'E' in the string.

These tests were broken out from the previous pattern matching examples because they failed on so many tests.

Pattern Matching a Number

Description: This is a continuation of the previous examples but with a more "real world" application

Example:

```
ISANUM='?(-)+([0-9])'

NNUMBER=-45
NUMBER=88
STRING=abc

if [[ ${NUMBER} = ${ISANUM}
]]
then
  echo PASS
fi
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	√	√	X	√	X	√
ksh	√	√	X	√	X	√
ksh93	∅	√	√	√	√	∅
bash	√	∅	X	√	√	∅

```
if [[ ${NNUMBER} = ${ISANUM}
]]
then
  echo PASS
fi

if [[ ${STRING} != ${ISANUM}
]]
then
  echo PASS
fi
```

Expected Results:

```
PASS
PASS
PASS
```

Notes: This test does not relate much that the previous two tests did not, it is simply a more practical application of the test. It did fail to detect two numbers on Solars ksh that had passed other pattern matches in previous examples. This may be resolvable by tweaking the regular expression.

Korn Style Functions

Description: This is the new style of (Korn) function declaration

Example:

```
function ksh_style_func
{
    echo ${1}
}

ksh_style_func PASS
```

Expected Results:

```
PASS
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	√	√	X	√	X	√
ksh	√	√	√	√	√	√
ksh93	∅	√	√	√	√	∅
bash	√	∅	√	√	√	∅

Notes: Bourne functions are declared using "function_name ()" syntax. Korn introduced "function function_name". Both are supported by Korn and Bash. Strict Bourne implementations do not support the Korn style declaration.

Proper Korn 93 functions have scope of variables as well as signals. Some of this is investigated in the next (Recursive Functions) example. The test here is if we can use the Korn style function declaration method or is it necessary to use both Korn and Bourne.

As we can see from this test, Korn style functions are supported virtually everywhere. The next test will demonstrate that not all aspects of Korn functions are supported.

Recursive Functions

Description: This is a test of recursion / variable scoping in the shell

Example:

```
function helper_func
{
  echo ${cnt}
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	X	X	X	X	X	X
ksh	√	X	X	√	X	X

ksh93	∅	√	X	√	√	∅
bash	X	∅	X	X	X	∅

```
}

function recursive_func
{
  typeset cnt=$(( $1 - 1 ))
  helper_func

  if (( ${cnt} == 0 ))
  then
    echo PASS
  else
    recursive_func ${cnt}
  fi
}

cnt=4
recursive_func ${cnt}
```

Expected Results:

```
4
4
4
4
PASS
```

Notes: The cnt variable here is declared local to the recursive function, but does not need to be for the "recursion" to appear to work in most shells. The cnt variable can be accessed as a global and the recursion will still count down to zero using that global (provided nothing else were to access that variable). The problem is that recursion using a global is not reentrant. This script tends to run on all shells tested, the key difference is scoping and what the helper_func prints.

Because helper_func does not have a local version of cnt in its scope it will print the global value (that never changes). A "broken" implementation will decrement the global (because it does not handle the local scope) and will cause helper_func to print a decrementing list of numbers instead of the expected (constant) global value.

Typically I use a case-sensitive method of denoting local / global variables, where globals (typically those inherited from the environment or calling / wrapper scripts) are in all caps while locals are all lower case. In this example that convention was not used to *force* the namespace collision between the local and global versions of cnt.

Also of interest is that while Bash supports the "local" declaration and seems to properly scope the variable when using the local syntax, it does so *differently*. In the Bash scoping rules the local is honoured in the recursive_func but apparently inherited by the helper_func. At the end of the run, the global value does not change (as one would expect) but the value printed by helper_func is different for the two scoping examples. I think the takeaway is to use recursion *carefully* and to never assume reentrancy because it *works*.

Numeric Tests

Description: Use C-style numeric evaluations and "truth" results

Example:

```
X=1
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	√	√	X	√	X	√
ksh	√	√	√	√	√	√

ksh93	∅	√	√	√	√	∅
bash	√	∅	√	√	√	∅

```
if (( 2 > ${X} ))
then
  echo PASS
else
  echo FAIL
fi
```

Expected Results:

```
PASS
```

Notes: This is a simple numeric test. It is not a Bourne convention, but it passes nearly everywhere.

Another point this example illustrates is the difference between the test value and the return value of the (( )) operation. *Programmatic* truth is a non-zero value, while *shell* truth is 0. The shell "if" keyword develops a sense of truth from *good* return values of 0. A clearer example of this is using the simple ((${X})) test where the value of X is 1 yet the conditional passes with the return value of 0.

C-Style Iteration

Description: Iterate in a C style / syntax "for" loop

Example:

```
for (( i=0 ; i < 4 ; i++ ))
do
  echo PASS
done
```

Expected Results:

```
PASS
PASS
PASS
PASS
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	√	X	X	√	X	X
ksh	√	X	X	√	X	X
ksh93	∅	√	√	√	√	∅
bash	√	∅	√	√	√	∅

Notes: I found a variant of this example somewhere on the Internet. I included it here only because I was curious where *non-traditional* code like *that* would work. Although the test does not relate the effects of accessing the counter in the loop, it does suggest that support is greater than one might initially expect.

print Command (With EOL Suppression)

Description: Is the print command availible in the shell

Example:

```
print -n PA
print SS
```

Expected Results:

```
PASS
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	X	√	X	X	X	√
ksh	√	√	√	√	√	√
ksh93	∅	√	√	√	√	∅
bash	X	∅	X	X	X	∅

Notes: This is the recommended method of text output to the user in Korn. The argument is that echo is not portable, which is true (see the following examples), but print tends not to be portable beyond Korn implementations. The result is effectively another output mechanism that makes portability difficult.

echo -n EOL Suppression

Description: description

Example:

```
echo -n PA
echo SS
```

Expected Results:

```
PASS
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	X	X	X	√	√	X
ksh	√	X	X	√	√	X
ksh93	∅	X	X	√	√	∅
bash	√	∅	√	√	√	∅

Notes: The -n option to echo tends to be a Linux / Bash convention. (Contrast with \c below.)

echo \c EOL Suppression

Description: description

Example:

```
echo "PA\c"
echo SS
```

Expected Results:

```
PASS
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	√	√	√	X	X	√
ksh	X	√	√	X	√	√
ksh93	∅	√	√	X	X	∅
bash	X	∅	X	X	X	∅

Notes: The \c method seems to be an everybody else convention. (Contrast with -n above.)

Compatible EOL Suppression

Description: EOL supression that will always work

Example:

```
printf "Testing for EOL
suppression..."
N=
C=
if `echo "X\c" | grep c >
/dev/null 2>&1`
then
  N=-n
  C=
else
  N=
  C='\c'
fi
printf "Done.\n"

echo $N "Testing EOL suppression
method.$C"
echo $N ".$C"
echo $N ".$C"
echo ".Done."
```

Expected Results:

```
Testing for EOL
suppression...Done.
Testing EOL suppression
method....Done.
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	√	√	√	√	√	√
ksh	√	√	√	√	√	√
ksh93	∅	√	√	√	√	∅
bash	√	∅	√	√	√	∅

Notes: When possible, I prefer to write "chatty" scripts that are self-documenting to the reader/maintainer as well as to the user. Instead of filling the script with comments I use output to relate what is going on in the script.

Specifically, I wrap an action with "Action to be performed..." and "Result." output. For this to format properly the EOL needs to be suppressed on the first output statement.

The above example uses the printf command to fill in while it figures out what the EOL suppression method is, then it uses the safer echo (that tends to be a) built-in.

Simple Math

Description: A very basic use of math in the shell

Example:

```
X=1

Y=$(( 2 + ${X} ))

Z=$(( 2 + Y ))

echo ${Z}
```

Expected Results:

```
5
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	√	√	X	√	√	√
ksh	√	√	√	√	√	√
ksh93	∅	√	√	√	√	∅
bash	√	∅	√	√	√	∅

Notes: There are numerous methods for math in shell scripts. The above method is fairly straight forward, efficient, and supported in all but the strictest Bourne implementations.

Floating Point Math

Description: Floating point math / floating point variables / calculating a percentage

Example:

```
typeset -lE N
typeset -lE D
typeset -lE R

N=1
D=3
R=$(( N / D ))
R=$(( R * 100 ))

printf "%2.0f%%\n" ${R}
```

Expected Results:

```
33%
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	X	X	X	X	X	X
ksh	√	X	X	√	X	X
ksh93	∅	√	√	√	√	∅
bash	X	∅	X	X	X	∅

Notes: Bash supports typeset for Korn compatibility, but prefers the declare keyword. The typeset syntax is used as builtin / internal floating point math is a Korn 93 thing (unless using and external such as bc).

Some ksh implementations set up aliases for "float". The float convention was less popular than the integer alias (used on another test). It was for this reason that I did not use float in the test.

Indexed Array

Description: A 0-based indexed array

Example:

```
typeset -a iarray

iarray[0]=P
iarray[1]=A
iarray[2]=S
iarray[3]=S

for i in 0 1 2 3
do
  echo ${iarray[$i]}
done
```

Expected Results:

```
P
A
S
S
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	√	X	X	√	X	X
ksh	√	X	X	√	X	X
ksh93	∅	X	X	√	√	∅
bash	√	∅	√	√	√	∅

Notes: This code for an indexed array failed primarily because of the explicit array typeset declaration. When the typeset line was not used, the test passed on all shells supporting Korn syntax (including Bash) - In other words, the same code without the explicit typeset passed on everything but the strictest Bourne implementations.

Associative Array

Description: An associative array can use strings as opposed to a 0 based index

Example:

```
typeset -A aarray

aarray[pee]=P
aarray[ahh]=A
aarray[ess]=S
aarray[sss]=S

for i in pee ahh ess sss
do
  echo ${aarray[$i]}
done
```

Expected Results:

```
P
A
S
S
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	X	X	X	X	X	X
ksh	√	X	X	√	X	X
ksh93	∅	√	√	√	√	∅
bash	X	∅	X	X	√	∅

Notes: This test does pass on the latest versions of Bash (fresh FreeBSD port) but does not appear to be standard on anything other than Korn 93 implementations.

Get Array Keys

Description: Return only the keys of an array

Compatibility:

Example:

```
myarray=( [PASS]="pass" )

echo ${!myarray[*]}
```

Expected Results:

```
PASS
```

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	X	X	X	X	X	X
ksh	√	X	X	√	√	X
ksh93	∅	√	√	√	√	∅
bash	X	∅	X	X	X	∅

Notes: This test is simplified to a single member array - primarily to insure that it did not fail on ordering of the results.

Command Substitution

Description: Test for $(...) instead of \`...\` style command substitution

Example:

```
TMPFILE=/tmp/deleteme.$$

echo "PASS" > ${TMPFILE}

echo $(< ${TMPFILE})

rm ${TMPFILE}
```

Expected Results:

```
PASS
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	√	√	X	√	X	√
ksh	√	√	√	√	√	√
ksh93	∅	√	√	√	√	∅
bash	√	∅	√	√	√	∅

Notes: The recommended reason for using this convention over \`...\` is that it does not require special treatment of some characters.

An additional note is the use of $(< file) as opposed to $(cat file). The former is handled by the shell, the latter by an external helper app.

Korn 93 Command Substitution

Description: Test for ${...} style command substitution

Example:

```
TMPFILE=/tmp/deleteme.$$

echo "PASS" > ${TMPFILE}

echo ${ < ${TMPFILE} }

rm ${TMPFILE}
```

Expected Results:

```
PASS
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	X	X	X	X	X	X
ksh	X	X	X	X	X	X
ksh93	∅	X	X	X	√	∅
bash	X	∅	X	X	X	∅

Notes: The key difference of the ${ ... } syntax is that it does not spawn a sub-shell to run the command(s). This results in a potential speed improvement and the possiblity of sideffects in the current shell.

Compound Variables

Description: A compound variable is a struct-like variable

Example:

```
CMPVAR=( typeset PF="PASS"
         typeset -i VAL=1 )

if (( ${CMPVAR.VAL} == 1 ))
then
  echo ${CMPVAR.PF}
fi
```

Expected Results:

```
PASS
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	X	X	X	X	X	X
ksh	√	X	X	√	X	X
ksh93	∅	√	√	√	√	∅
bash	X	∅	X	X	X	∅

Notes: The primary value of such a variable (to me) is the ability to pass a *glob* of data as a single variable. Below is an example to pass a compound variable by reference. Obviously this is a Korn 93 concept.

```
function ShowResults
{
  typeset -n cmpvar=$1

  if (( ${cmpvar.VAL} == 1 ))
  then
    echo ${cmpvar.PF}
  fi
}

CMPVAR=( typeset PF="PASS"
         typeset -i VAL=1 )

ShowResults CMPVAR
```

Type Variables

Description: Create a "type" builtin that can be used to declare variable of this type.

Example:

```
typeset -T mystruct=( a= b= c=
)

mystruct ms

print ${@ms}
```

Expected Results:

```
mystruct
```

Compatibility:

	OS X	AIX	Solaris	Linux	FreeBSD	HP-UX
sh	X	X	X	X	X	X
ksh	X	X	X	X	X	X
ksh93	∅	X	X	X	√	∅
bash	X	∅	X	X	X	∅

Notes: The example here declares a type called `mystruct` that is used to declare a (compound) variable called "ms". The final line uses the `${@`*`varname`*`}` syntax to determine the type of the variable.

Shell Reference

Here are some sources for the above concepts:

Bourne	-	http://www.freebsd.org/cgi/man.cgi?query=sh&format=html
Korn 88	-	http://www.research.att.com/sw/download/man/man1/ksh88.html
Korn 93	-	http://www.research.att.com/sw/download/man/man1/ksh.html
Bash	-	http://www.gnu.org/software/bash/manual/bashref.html
QuickSheet	-	http://www.tablespace.net/quicksheet/shell-quicksheet.pdf

Some Notes

The practice of (hard or soft) linking one shell to another allows a OS distribution to support multiple common interpreter specifications (shebangs) without shipping / maintaining multiple shells. So this works well for the maintainer of the OS, but carries the risk of an administrator who then mistakenly believes that Bourne shell is truly as flexible as Korn or Bash. This paper should serve to address that misconception.

It must be noted that Solaris (and to a lesser extent FreeBSD) goes to such lengths to honor shell compatibility and standards. The shells packaged with Solaris are chosen with some thought as to script compatibility. This may have something to do with the heavy reliance on scripting by Solaris administrators. (Try running "file" on the which command.) It is believed that Sun will ship a Korn 93 version of ksh (as opposed to the dtksh "hack") with Solaris 11.

The tests were created using a shell based test suite. (Copies are available upon request.) Sample snippets and expected results files were created and run under each shell by pre-pending the shell magic / interpreter string to the test script and running it and comparing the actual to the expected results. The unfortunate side effect of this is that some tests "failed" when a minor change in the syntax would cause them to pass. This is most apparent with the typeset builtin. I have tried to acknowledge these *false* failures when they were encountered.

This is an evolving document.

By: William Favorite <wfavorite@tablespace.net>
Version: 0.24.2
Date: 9/28/10