# Jörg Schilling

# Bourne Shell, Korn Shell, POSIX, ...

# Unterschiede

# Fokus Fraunhofer

# Alte Shells unter UNIX

- Um 1970: Thompson Shell – der Vater aller Shells
  - If … goto als externe Kommandos, mit lseek(0)
  - Keine Variablen $0, $1, …
  - /etc/glob für wildchart Zeichen in Filenamen
- 1975 Mashey Shell als Patch auf Thompson Shell (osh)
- 1976 erste Version des Bourne Shell auch als osh Patch
  - Shell als Programmier-Sprache
  - Variable Subst., Signal Mgt., control flow
- Um 1977 csh auch auf Basis vom Thompson Shell

# Shells auf Bourne Shell-Code Basis

- Bourne Shell seit 1976, erste extern verteilte Version 1979
- Bourne Shell 1981 #, [!...], ${var:[-+?=]val}, set
- Bourne Shell 1983 shift
- Bourne Shell 1984 Funktionen, unset/echo/type, „buitlin redirect“
- Bourne Shell 1986 $@, 8-bit-clean, getopts, rekursive Funktionen
- Bourne Shell 1989 Job-control wie bei csh
- Bourne Shell 2006 *Schily Bourn Shell* (bosh) viele Erweiterungen
- Korn Shell 1983 auf Basis der Bourne Shell Source
- Korn Shell 1986 ksh86 mit eingebautem History Editor
- Korn Shell 1988 erweiterte Version ksh88, erkennbare Änderungen

# Shells auf BourneShell Basis

- **Korn Shell 1993 Neue Version ksh93 weitere Änderungen**
- **Korn Shell 1997 erste Version mit verfügbarer Source**
- **Korn Shell 2001 erste echte OpenSource ksh93 Version**
- **Dtksh Korn Shell '93 mit CDE Builtins kann GUI als Shell skript bauen**
- **Tksh Korn Shell '93 mit TCL/TK Builtins kann GUI als Shell skript bauen**

# Shells auf Basis der BourneShell Syntax

- Bash 1989
- Ash (Almquist Shell) 1989 *BSD Ersatz für BourneShell
  - *BSD hatte nie eine Lizenz für einen modernen Bourne Shell mit Funktionen
- Dash (Debian Almquist Shell) 2004 auf Basis von ash
- Pdksh (Public Domain ksh) 1988
- MirKorn Shell „mksh“ (auf Basis von pdksh) 2003
- Posh (Debian) pdksh Variante 2016
  - Geht nur mit glibc Bugs, unbenutzbar auf POSIX Zertifiziertem BS
- Zsh (als „zsh“ nicht Bourne Shell / POSIX kompatibel)
  - Mit Argv[0] == „sh“ POSIX-ähnlich

# Shells auf Basis der BourneShell Syntax

- **Busybox von Bruce Perens Viele Builtins, Bourne Shell ähnlich auf Basis von ash, entstand aber erst 1999 - nach ksh93, der bereits 1993 viele Builtins hatte**

# Unterschiede Bourne Shell, bosh, ksh

- Init Skripte, z.B. $HOME/.kshrc für interaktive Shells
- Pipe Aufbau Reihenfolge und Eltern Kind Verhältnis
- Variablen Zuweisungsreihenfolge
- Variablen Zuweisung vor Buitlins (Lebensdauer)
- Syntax-Fehler Behandlung bei Builtins
- History Editor
- Aliases
- Umask Parameter -S / chmod like Mask
- $PWD und andere Variablen

# Unterschiede Bourne Shell, bosh, ksh

- ${.sh.*} Variablen
- Eingebaute Zeitmessungen
- pushd/popd/dirs (nur bosh und bash, zsh)
- dosh  Builtin für parameterisierte Aliase (nur bosh)
- export/readonly Features
- Killpg (nur bosh)
- set -o longopt (alle POSIX Shells)
- test Features
- ~  (Tilde Expansion)

# Basis-Features Bourne Shell

- for *name* [ in *word* ... ] do *list* done
- case *word* in [ *pattern* [ | *pattern* ] ) *list* ;; ] ... esac
- if *list* then *list* [ elif *list* then *list* ] ... [ else *list* ] fi
- while *list* do *list* done
- until *list* do *list* done
- (*list*)
- { *list*;}
- *name* () { *list*;}

# Syntax-Erweiterungen im ksh

- for (( [*expr1*] ; [*expr2*] ; [*expr3*] )) ;do *list* ;done (nicht POSIX)
- select *vname* [ in *word* ... ] ;do *list* ;done (nicht POSIX)
- ((*expression*)) (nicht POSIX)
- [[ *expression* ]] (nicht POSIX)
- time [ *pipeline* ] (nicht POSIX als Teil der Shell Syntax)
- ! *cmd* (auch POSIX)
- $(*cmd*) (auch POSIX)
- $((arithmetic *expression*)) (auch POSIX)

# Init Skripte

- **Bourne Shell: /etc/profile + ~/.profile bei login**
- **Bosh: /etc/profile + ~/.profile (login), /etc/sh.rc + /.shrc (i)**
- **Ksh93: /etc/profile + ~/.profile (login), /etc/ksh.kshrc + ~/.kshrc (i)**
- **Bei allen: kein .logout Skript, da es `trap cmd EXIT` gibt**

# Pipe Aufbau, Eltern Kind Verhältnis

- Kommandointerpretation nicht im Standard vorgegeben
- Einfaches Beispiel: `cmd1 | cmd2`
- Bourne Shell: cmd1 ist Kind von cmd2, cmd2 ist Kind von sh
- Ksh93/bosh: beide Kommandos sind Kinder vom Shell
- Beispiel mit Builtin: `cmd1 | read var`
- Bourne Shell: `read` im Kind von sh, `cmd1` ist Kind von read
- Ksh93: `read` läuft im ksh, `cmd1` ist Kind von ksh
- bosh: ähnlich wie ksh93
- Konsequenzen? Antwort aus dem Auditorium

# Variablenzuweisungen

- `var1=val1 var2=val2 cmd` Kommando mit temporären (privaten) Environment Variablen
- Im statischen Fall kein Problem...
- Aber: `var1=val1 var2=$var1 cmd`
- Beim Bourne Shell ist `var2` hier leer!
- Fix ist in ksh und bosh, POSIX sagt nichts dazu

# Temporäre Variablen + Builtin Kommandos

- `var=val cmd` Erzeugt Kind, das var zuweist und cmd ruft
- `CDPATH=val cd` naheliegend gleich? aber problematisch
- „`cd`“ ist ein eingebautes Kommando und muß es sein
- Die Variablenzuweisung erfolgt daher im Shell selbst
- Beim Bourne Shell wirken alle Variablen vor allen Builtin-Kommandos dauerhaft auf den ganzen Shell nach
- Bei ksh und bosh wird dies bei den meisten Kommandos verhindert. Ausnahme: POSIX „special builtins“
- POSIX: bei „special builtins“ müssen vars in den Shell

# Nomenklatur zu Builtins

- Einige Kommandos müssen im Shell implementiert sein damit sie funktionieren (z.B. cd)
- Der usprüngliche Bourne Shell hatte wenige Builtins
- Später kamen viele dazu
- Unübliches Verhalten ist nicht akzeptabel für Kommandos, denen man nicht ansieht daß sie im Shell implementiert sind (z.B. das ksh-builtin `uname`)
- POSIX führt „special builtins“ ein, anderes Verhalten OK
- Liste der „special builtins“ enthält nicht cd, alias, …
- Neu SUSv7 tc2: „intrinsic commands“ ohne anderes Verhalten

# Syntax Fehlerbehandlung bei Builtins

- Änderungen am Shell um das Verhalten verständlich zu machen: Externe Kommandos <-> Builtins
- Bei externen Kommandos führt jeder Fehler maximal zu einem Exit Code != 0
- Klassische Methode im Bourne Shell: fatale Fehler in Builtins führen bei interaktivem Shell zu longjmp() vor Prompt-Ausgabe, sonst (wenn nicht interaktiv) zu exit des Shells
- Ksh: weitgehende Angleichung der Builtins an externe Kommandos – außer „special builtins“
- Bosh: ähnlich wie ksh
- POSIX: „`command` <special-builtin>“ kein spezielles verh.

# History

- Der ursprüngliche Bourne Shell hat keine History
- Ksh: ab ca. 1981 Ursprünglich nur Builtin Erweiterungen zur Listenverarbeitung im Bourne Shell, daher zunächst auch keine History
- Ksh ab ca. 1983 History mit Hilfe des Kommandos `fc` editierbar, aber nur durch extern gerufenen `ed/vi`.
- Später ab ca. 1986: eingebauter History Editor
- Schily Bourne Shell (`bosh`) ab 2006 mit dem History Editor des bsh (H. Berthold AG) Konzept/Prototyp in TU-Bln 1982, erste Implementierung unter UNOS in `bsh` 1984. Emuliert: `ved` (UNOS Editor)

# Aliase

- Bourne Shell ursprünglich ohne Aliase
- Ksh führt um 1984 Aliase ähnlich zum csh ein
- Schily Bourne Shell: ab 2012 Aliase ähnlich dem Kommando Interpreter von UNOS (1980) mit persistenten Alises und directory-spezifischen Aliases (POSIX++)
- Ksh: Parameterisierte Aliase wie beim csh sind nicht möglich
- Bosh: Parameterisierte Aliase mit Hilfe des Builtins „`dosh`" das minimale-Shell-Skripte wie mit `sh -c cmd` emuliert ohne jedoch einen neuen Shell zu starten

# export/readonly

- Der Bourne Shell versteht `export VAR` bzw. `readonly VAR`
- Ksh, bosh und POSIX erlauben auch:
  - `export VAR=value ...`
  - `readonly VAR=value ...`
- sowie:
  - `export -p`
  - `readonly -p`
- Um eine Ausgabe zu bekommen, die wieder als Shell-Kommando nutzbar ist
  - `export -p > file`
  - `. file`
  - Kann aber aber keine Variablen in den „unset“ Status zurückführen

# umask

- Das builtin „`umask`“ im Bourne Shell versteht nur oktale Werte im Argument „Maske“
  - Natchteil: Die „Maske“ ist das Inverse der Zugriffs-rechte und daher schwer verständlich
- Ksh führt `umask -S` ein, Ausgabe wie `chmod`, ähnlich zu `ls`
- Bosh folgt dem Beispiel, denn das wird durch POSIX ge-fordert
- Auch `umask u=rwx,g=rx,o=rx` ist als Ersatz zu `umask 022` möglich

# Shell Variablen

- Der ursprüngliche Bourne Shell kennt nur wenige Shell-Variablen: SHELL, PATH, HOME, IFS, MAIL, MAILCHECK, MAILPATH, PS1, PS2, SHACCT
- Ksh führt viele neue Variablen ein. Einige werden in POSIX übernommen: ENV, LINENO, PPID, PS4, PWD
- Bosh übernimmt alle POSIX Erweiterungen
- Ksh und bosh: TIMEFORMAT für „time“ Formatierung
- Bosh: zusätzlich eingebautes Timing für alle Kommandos mit `set -o time` (ähnlich `csh` und `bsh`)

# Erweiterte Variablen

- Bourne Shell: Verwendung von Variablen die mit einem Punkt anfangen führen zu einem longjmp() vor den interaktiven Prompt, bzw. zum Exit des ganzen Shells
- Portable Nutzung mit Subshell: `(echo ${.sh.version})`
- Ksh und bosh: `${.sh.*}` Variablen für Sonderzwecke
- Bosh: Einführung von `.sh.*` um 32 Bit Exit Codes zu verarbeiten → man Page
- Alle anderen Shells: nur die unteren 8 Bit des Exit Codes der Kinder sind sichtbar obwohl es seit 1989 waitid() gibt und waitid() seit 1997 Bestandteil von POSIX ist

# Zeitmessungen

- Klassische UNIX Methode `/usr/bin/time kommando`
- Ksh implementiert „`time`“ als Teil der Syntax
- Bosh implementiert eingebautes Timing für alle Kommandos und „`time`“ als reserved Word wie ksh
- Ksh und bosh deaktivieren das eingebaute „`time`“, wenn `time` mit einer Option aufgerufen wird (z.B. `-p`)
  - Dadurch wird POSIX-Konformität erreicht

# Directory-Verwaltung

- Bourne Shell: nur „cd“ ist verfügbar und ohne Optionen
- Bourne Shell 1989 von Svr4: CDPATH Variable wird unterstützt, „cd“ ist immer „physisch“
- Ksh: ähnlich zum Bourne Shell von Svr4 aber „cd“ ist „logisch“ und Optionen -L / -P (default ist -L)
- Bosh: zusätzlich die Kommandos `pushd`/`popd` und `dirs`
  - Sie verwalten einen Directory „stack“
  - Optionen -L / -P (default ist -P wie Bourne Shell)
  - `bosh -o posix` → Default ist -L

# Parameterisierte Alises

- Csh versteht: `alias lm 'ls -l \!* | more'`
  - \!* wird durch die Argument Liste expandiert
- Bourne Shell kennt keine Aliase
- Ksh versteht nur einfache Aliase ohne Parameter
- Bosh kennt das Builtin „`dosh`“, das aus dem UNOS Kommmando Interpreter von 1980 abgeleitet wurde:
  - z.B: `alias lm='dosh '\''ls -l "$@" | more'\'' lm'`
  - Ein einzeiliges eingebautes Skript ermöglicht die Parameterisierung der Aliase

# Komplexe Aliase verständlich editieren

- Wie im Vorigen Beispiel ersichtlich: die ksh/POSIX Syntax für aliase kann unlesbar werden
- Bosh kennt daher ein Verfahren zur Bearbeitung von Aliases im „RAW“ Modus:
  - `alias -R` zeigt mit bosh Aliase im RAW Modus an
  - Unser Beispiel:

```
#pb lm          dosh 'ls -l "$@" | more' lm
```

  - Wird plötzlich lesbar
- Zum Editieren: `set -o hashcmds` einschalten
- Dann `#lh lm` eingeben und Cursor hoch....

# Lange Optionen

- Der Bourne Shell kennt nur wenige Optionen wie `-v`
- Ksh führt lange Optionen ein und POSIX übernimmt
- Auflisten mit: `set -o / set +o`
- Statt `set -v` einschalten nun auch mit `set -o verbose`
- Bosh übernmmt dies, da es in POSIX ist

# Lange Optionen

- Bosh kann auch lange Optionen im eingebauten getopts(1)
  - z.B. `--lang` als Alias zu `-l` mit optstring: „l(lang)“
  - aber auch `--lang` ohne entsprechende kurze Option
  - und `-lang`, wenn „optstring“ mit „()“ anfängt
- Ksh93 unterstützt die erste Variante
- Bash unterstützt keine langen Optionen mit getopts(1)

# Test Features

- Da „`test`“ seit Mitte der 1980er ein eingebautes Kommando ist, bestimmt der Shell die „`test`“ Features
- Bourne Shell hat kaum neue Möglichkeiten eingebaut
- Ksh kennt vieles mehr (z.B. `-e file`) daher aufpassen wenn es portabel sein soll
- Bosh kennt alle ksh test Features (außer `-a file`)

# „test“ und POSIX

- Klassisches UNIX „test“ funktioniert wie ein UNIX-Kommando mit Optionen. Das ist problematisch:
  - Einzelne Argumente sind meist Shell Variablen
  - Dadurch unvorhersehbare Argumente
  - „test“ kann daher nicht erkennen ob ein Argument ein Oparator oder ein Parameter ist
  - Komplexe Ausdrücke können daher zu unerwarteten Ergebnissen führen

# POSIX „test“

- POSIX behandelt „`test`“ nur nach Anzahl der Parameter
- Kein Argument → exit != 0
- Ein Argument → exit 0 wenn strlen(arg) > 0
- Zwei Argumente
  - ! + Ein Argument
  - Unary Oparator + Argument
- Drei Argumente
  - Argument + binary op + Argument
  - ! + Zwei Argumente
  - ( Ein Argument )
- Vier Argumente
  - ! + Drei Argumente
  - ( Zwei Argumente )

# „test“ portabel machen

- Komplexe Ausdrücke vermeiden
- Eindeutige Ausdrücke verwenden
- Komplexe Ausdrücke in mehrere Einfache zerlegen:
  - [ $a -gt $b -a -f $f ] umwandeln in →
  - [ $a -gt $b ] && [ -f $f ]
  - Mehrere „test“ Aufrufe sind kein Performance Problem da built-in

# Abkürzungen für das Heimatverzeichnis

- Bourne Shell kennt nur `$HOME`
- Ksh führt zusätzlich (wie csh) ~ ein
  - `~/` → Eigene Homedirectory
  - `~joe/` → Homedirectory vom User joe
  - `~+` → Aktuelles Verzeichnis ($PWD)
  - `~-` → Voriges Verzeichnis ($OLDPWD)
- Bosh implementiert dies wie ksh
- Ksh und bosh: TAB expandiert ~ im Editor

# Ksh for- Schleife

- **Die ksh Erweiterung `for ((i=0; i < 10, i++))  ; do list; done` ist zwar im Bash verfügbar aber nicht POSIX**
- **Daher auch nicht in Bosh**

# Select Kommando

- Das Select Kommando des ksh ist auch in vielen anderen „ksh-alike“ Shells, aber nicht in POSIX
- Bourne Shell kennt es nicht
- Bosh kennt es wie ksh, denn es ist einfach zu implementieren und effizient
- Achtung: dash (/bin/sh auf Linux) kennt es nicht

# Arithmetische Ausdrücke

- `((expression))` ist in ksh und bash aber nicht POSIX
- Bourne Shell kennt es nicht
- Bei POSIX kompatiblen Shells gibt es:
  - `$((expression))`
- Achtung:
  - `((expression))` von ksh ist Teil der Syntax, kann aber nur anstelle von Kommandos stehen: if ((ex))
  - `$((expression))` ist eine Art Makro-Expansion
  - POSIX: $((expression)) oder $( (cmd)), ksh erkennt es am Zusammenhang (Arithmetik vs. Sub-Shell)

# Test Erweiterungen

- `[[expression]]` ist in ksh und bash aber nicht POSIX
- Bourne Shell kennt es nicht
- Bei POSIX kompatiblen Shells und Bourne Shell gibt es:
  - „test“ und „[“
- Achtung:
  - `[[expression]]` in ksh ist Teil der Syntax
  - `[ expression ]` ist ein eingebautes Kommando

# ! cmd

- `! cmd` Negiert den Exit Code (POSIX)
- Daher in vielen Shells verfügbar
- Nicht im Bourne Shell
- Aber in Bosh

# Kommando Ersetzung

- Bourne Shell kennt: `` `cmd` `` um in der Kommandozeile die Ausgabe von „cmd“ einzubauen
- Das ist aber geschachtelt ähnlich komplex wie die ksh Alias-Definitionen
- Ksh führte daher `$(cmd)` ein und POSIX übernimmt es
  - Das ist allerdings sehr schwer zu parsieren und benötigt einen rekursiven Parser zum Auffinden des Endes
  - Korrekt implementiert nur in ksh93, bosh, bash-4, mksh

# Arithmetische Ersetzung

- Nicht verfügbar im Bourne Shell
- Ksh: $((expression)) in der Kommandozeile wird ersetzt
- Auch verfügbar in bosh
- Warnung Leerzeichen zwischen den Klammern nötig, falls eine Kommandoersetzung in einem Subshell gemeint ist
- Ksh macht dazu Plausibilitätsbetrachtungen aber das ist nicht portabel und ksh kann nicht alles finden:
  - $((ls)) ist ein gültiger arithmetischer Ausdruck! (0)

# Das eingebaute `read` Kommando

- Das „read“ Kommando liest eine logische Zeile, teilt sie an `IFS` Zeichen und füllt die Ergebnisse in Variablen
- Der Bourne Shell liest zuerst beliebig viele `IFS` Zeichen am Zeilenanfang, dann ein oder mehrere Felder die durch beliebig viele `IFS` Zeichen getrennt sind
- POSIX macht es genauso wenn `IFS` nur „spaces“ enthält
- POSIX erzeugt leere Variablen bei wiederholten „non-space“ Zeichen aus `IFS`
- POSIX erlaubt daher /etc/passwd mit „read“ zu lesen:
  - `IFS=: read NAME PWD UID GID GCOS HOME SHELL < /etc/passwd`

# „Lokale“ Variablen in Funktionen

- Bourne Shell Funktionen „`f() { cmd; }`“ wie in POSIX
- Ksh kennt 2 Typen von Funktionen:
  - POSIX Funktionen: „`f() { cmd; }`“
  - Ksh-like: „`function f { cmd; }`“
- Für sinnvolle Rekursion werden „lokale“ Variablen benötigt
- Bourne Shell und POSIX kennen keine „lokalen“ Variablen
- bash, bosh, dash, ksh88, mksh kennen „`local`“
- Ksh93 kennt lokale Variablen nur in ksh-Funktionen mit „`typeset`“

# Weitere ksh Erweiterungen

- Als der Bourne Shell 1979 entstand, gab es kein Pattern-Matching in libc
- Der Bourne Shell verwendet die eigene Funktion „gmatch()“ die „*“, „?“, „`[...]`“ und „`[!...]`“ unterstützt
  - Verwendet wird sie bei Dateinamen und bei `case`
- Ksh verwendet inzwischen teilweise Regular expressions, z.B. mit `[[ $v1 =~ $v2 ]]`
- Ksh93 unterstützt „**“ für einen ganzen Directorybaum, wenn die Shell option -G (auch set -o globstar) gesetzt ist
- Ksh kennt viele Stringverarbeitungsfunktionen

# Weitere ksh Erweiterungen

- Ksh kennt viele Stringverarbeitungsfunktionen:
  - ${parameter%word} Remove smallest suffix
  - ${parameter%%word} Remove largest suffix
  - ${parameter#word} Remove smallest prefix
  - ${parameter##word} Remove largest prefix
  - ${#parameter} Anzahl der Buchstaben in parameter
  - Alle obigen Parameter-Ersetzungen sind in POSIX und bosh
  - Ksh kennt weitere non-POSIX Funktionen für sub-strings

# Weitere ksh Erweiterungen

- Ksh unterstützt Array Variablen
- Ksh unterstützt „here“-Worte
  - cat <<<word
- Ksh unterstützt „Process Substitution“
  - cat <(command)
- Ksh unterstützt Netzwerk-IO
  - cat </dev/tcp/*host*/*port*
- Ksh unterstützt Ko-Prozesse

# Weitere Unterschiede Bourne Shell / POSIX

- ^ ist im Bourne Shell ein Alias für `|` und muß sonst gequotet werden, bei POSIX ist das nicht nötig
  - Bosh schaltet das mit „set -o posix“ um
- Im Bourne Shell sind `${var:+ }` zwei Argumente
- POSIX verlangt `${...}` als Block zu parsieren
- `${var:+\ }` funktioniert immer
- Bourne Shell ruft mit `IFS=o; violet` den `vi` mit arg `let`
- POSIX macht „field Splitting“ nur in Makro Expansionen
- Bourne Shell macht „field Splitting“ überall

# Weitere Unterschiede Bourne Shell / POSIX

- POSIX erzeugt bei „field Splitting“ mit wiederholten non-blank Zeichen leere Felder (siehe auch „read“ Kommando)
  - Daran erkennt man daß „dash“ keine multi-byte chars kann
- Bourne Shell ignoriert alle leeren Felder.
- Bourne Shell: Der Exit Code von „`case`“ ohne Match ist der Exit Code des letzen Kommandos, das ist verwirrend
- Der Exit Code von „`case`“ ohne Match bei POSIX ist 0
- Portabler Code sollte „ : “ vor das „`case`“ schreiben

# Weitere Unterschiede Bourne Shell / POSIX

- **Bourne Shell importiert Variablen aus dem Environment**
  - **Dabei werden die Werte ins Env. der Kinder weitergereicht**
  - **Interne Shell-Variablen mit gleichem Inhalt werden erzeugt**
  - **Die Internen Variablen können unabhängig verändert werden**
  - **Erst „`export var`“ exportiert die modifizierten Werte**
- **POSIX importiert Variablen aus dem Environment**
  - **Dabei werden importierte Variablen automatisch exportiert**
  - **Veränderte Shell Variablen werden direkt mit neuem Wert exportiert**
- **Bosh: wie Bourne Shell, mit „`set -o posix`“ POSIX**

# Weitere Unterschiede Bourne Shell / POSIX

- Der Bourne Shell erlaubt kein `unset` auf die Variablen `IFS PATH PS1 PS2 MAILCHECK`
- POSIX erlaubt ein `unset` auf alle Variablen
- Der Bourne Shell erlaubt kein `trap` auf `SIGSEGV`
- Der Bourne Shell initialisiert `PWD` nicht zum Start
- Der Bourne Shell macht keine Parameter Expansion mit `PS1` und `PS2`
- Der Bourne Shell kennt nur `$0` .. `$9`, POSIX auch `$99`
- Der Bourne Shell löscht die Argumente nicht mit `set --`
- `shift $#` geht jedoch mit allen Shells

# Weitere Unterschiede Bourne Shell / POSIX

- Der Bourne Shell hat seit 1981 einen Bug mit

  ```
  continue <große Zahl>
  ```

  und macht dann „`break`“ statt „`continue`“
- Der Bourne Shell hat seit 1981 einen Bug mit

  ```
  cat 0<<-EOF
  ```

  und strippt dann keine TABs
- Der Bourne Shell hat seit 1977 einen Bug mit

  ```
  for i in 1 2 3; do echo $i; break 0; done
  ```

  und erstarrt wenn er dabei im interaktiven Modus ist

# Weitere Unterschiede Bourne Shell / POSIX

- Der Bourne Shell erlaubt nur ein Optionsargument, „-vc“
- POSIX verlangt „normale“ Options-Bearbeitung
  - `sh -x -m` funktioniert daher
- Der Bourne Shell definiert bei:

  ```
  sh -c command
  ```

  - „`command`“ als Argument der Option `-c`
- POSIX definiert `-c` als normale Option die an beliebiger Stelle (vor den Argumenten) stehen darf, also z.B.:
  - `sh -c -o noglob 'echo *'`

# Weitere Unterschiede Bourne Shell / POSIX

- Der Bourne Shell beendet ein Skript mit dem Code:
  - `cd non-existent`
- Bei POSIX ist nur der Exit-Code von „`cd`“ != 0
- Portable Skripte verwenden immer: `cd $dir || exit`

# Experimentelle bosh Features

- Bosh hat „find“ als eingebautes Kommando
  - Das funktioniert auf Basis von“libfind“
  - Daher läuft „find“ direkt im bosh (ohne fork()/exec())
  - Daher könnte „find“ auch Code aus bosh rufen
  - Libfind hat seit August 2018 ein „-call“ das dies tut
    - Wie eval(1) wenn kein „$“ im ersten Argument
    - Wie dosh(1) (sh -c) wenn „$“ im ersten Argument
  - „-call“ ist 20-40x schneller als „-exec“

# Übung 1

- **Erklären Sie warum**
  - `$shell -c 'date | read VAR; echo $VAR'`
- **Nur bei $shell == ksh, bosh und zsh das Datum ausgibt, nicht aber bei bash, sh und dash**
- **Entwickeln Sie ein Kommando daß das Datum bei allen Shells der Variable VAR zuweist**

# Übung 2

- **Das Kommando `VAR1=bla VAR2=$VAR1 /usr/bin/env` listet eine leere Variable VAR2 wenn ein klassischer Bourne Shell verwendet wird. Warum passiert das und gibt es eine Lösung mit der die erwartete Ausgabe bei allen Shells nutzbar wird?**

# Übung 3

- **Welchen Wert hat die Variable FOO nach Ausführen des Kommandos**

```
FOO=bar pwd
```

- **In den diversen Shells?**
- **Warum unterscheiden sich die Ergebnisse?**

# Übung 4

- Lesen Sie die Man Pages von Bourne Shell, bash, ksh, bosh und suchen Sie nach einer Möglichkeit bei Expansion der Variablen FOO folgenden Text zu erzeugen:
  - „gesetzt“ nur wenn $FOO nicht leer ist, sonst $FOO
  - „leer“ nur wenn $FOO leer ist, sonst $FOO
  - „fehlt“ nur wenn $FOO nicht existiert, sonst $FOO
- Hinweis: die Erklärung befindet sich im Abschnitt „Parameter substitution“
- Ist dies mit allen Shells möglich?
- Liefern alle Shells brauchbare Man Pages?

# Übung 5

- Das Kommando

```
sh -c 'test -R && echo bla'
```

  gibt „bla“ aus
- Erklären Sie warum
- Das Kommando

```
sh -c 'test 2 -GT 3; echo foo'
```

  gibt mit einigen Shells „`foo`“ aus, aber nicht mit allen
- Erklären Sie warum

- **Das Kommando:**

```
sh –c 'exit 1234567890' ; echo $?
```

**Gibt `210` aus, Erklären Sie warum nicht `1234567890`**

- **Aus welchem Shell und unter welchen Umständen schaffen Sie das erwartete Ergebnis zu erzielen, wenn Sie dort das angegebene Kommando aufrufen?**
- **Mit:**

```
ksh –c 'exit 1234567890' ; echo $?
```

**klappt das unabhängig vom aktuellen Shell nie, woran könnte das liegen?**

# Übung 7

- **Schreiben Sie ein kleines ksh Skript, das ein Menü mit 3 Auswahlmöglichkeiten ausgibt.**

- **Schreiben Sie dieses ksh skript:**

```
farbe=braun-gelb-blau
if [[ $farbe == *gelb* ]]; then
        echo "mit gelb"
fi
```

- **So um, daß es auch mit dem Bourne Shell das gewünschte Ergebnis liefert**

# Antwort 1

- Bei ksh läuft das „`read`“ nicht in einem Subprozess
- Die Variable wird nur dort, nicht im Hauptshell verändert
- Das portable Kommando ist:
  - `sh -c 'VAR=`date`; echo $VAR'`

# Antwort 2

- Portabel kann man nur `VAR1` und `VAR2` auf separaten Kommandozeilen setzen
- Dann werden aber `VAR1` und `VAR2` Bestandteil des Haupt-Shell-Prozesses

# Antwort 3

- Bei allen Shells, die Builtins POSIX-konform implementieren, hat `VAR=val pwd` keine Auswirkungen auf den eigentlichen Shell-Prozess
- Grund: Historisch hat sich das immer auf den ganzen Shell ausgewirkt, es wurde aber mit ksh88 geändert und von dort in POSIX übernommen

# Antwort 4

- `FOO=irgendwas`
- `${FOO:+gesetzt}`
- `FOO=""`
- `${FOO:-${FOO+leer}}`
- `unset FOO`
- `${FOO-fehlt}`
- **Die Expansion ist mit allen Shells möglich wenn sie mindestens kompatibel zum Bourne Shell Stand 1981 sind**

# Antwort 5

- Weil „`-R`“ ein String ist, dessen Länge > 0 ist
- Weil `-GT` ein illegaler Operator ist gibt es beim historischen Bourne Shell ein exit() des ganzen Shells, bei POSIX Kompatibilität jedoch lediglich exit() mit Exit-Code != 0 für das test Kommando

# Antwort 6

- `210` ist `12345678980 & 0xFF`
- Wenn aus dem bosh mit „set -o fullexitcode“ gearbeitet wird und man auf einem modernen BS ist, klappt es
- Ksh maskiert schon den Parameter des eingenauten exit Kommandos mit 0xFF bevor das eigentliche `exit()` ausgeführt wird

# Antwort 7

```
select i in wer wo was; do
    echo Selektiert: $i  aus Nummer: $REPLY
    break
done
```

# Antwort 8

```
farbe=braun-gelb-blau
case $farbe in
*gelb*)
        echo “mit gelb“
        ;;
esac
```

# Probleme mit POSIX und dem Shell

- Normalerweise standardisiert POSIX das was am Markt ausreichend einheitlich verfügbar ist
- Es gibt aber viele Shell Implementierungen mit vielen zu einander inkompatiblen Erweiterungen
- Daher muß hier die Initiative von der Standardisierung ausgehen und Einfluß auf alle Shells nehmen
- Die normale Methode, Alles teilweise inkompatible als „unspecified“ zu bezeichnen wäre unbenutzbar
- Daher werden sinnvolle Shell Erweiterungen aufgegriffen und durch das POSIX Komitee neu formuliert

# Planungen für POSIX Issue 8

- Unterstützung für die ksh $'...' Expansion
  - C-ähnliche String Escape Darstellung, z.B. $'\n'
  - Unicode Zeichen mit $'\uxxxx'
- Reservierter neuer POSIX Namensraum
  - Auflisten der inkompatiblen Builtins
  - POSIX Flags für set(1)
  - Prüfen oder aktivieren von erweiterten Features
- 32 Bit exit Codes mittels `waitid()`
  - Vermutlich mit Hilfe einer neuen Shell-Variablen $/
- siginfo_t Werte in „trap“ Kommandos verfügbar machen

# Suchpfade für eingebaute Kommandos

- Alle Shells haben eigene Builtin-Kommandos
- Viele der Kommandos entsprechen normalen UNIX Kommandos mit leichten Abweichungen
- Normale Shell-Regel: ein Builtin hat immer Vorrang
- PATH ermöglicht bei separaten Binaries eine Auswahl
- Für die ksh93 Integration in OpenSolaris wurde PATH für Builtins gefordert und implementiert
- Bei ksh93 mit dem Kommando „`builtin`“:
  - PATH Zuordnung für jedes Builtin ändern
  - Einzelne Builtins deaktivieren oder reaktivieren

# Suchpfade für eingebaute Kommandos

- Die meisten ksh93 Builtins sind dem Pfad `/usr/ast/bin` zugeordnet.
- `/usr/ast/bin` ist nicht im normalen PATH
- Einige wenige Kommandos sind `/usr/bin` zugeordnet
  - Allerdings nur auf Solaris
  - Ein solches Kommando auf Solaris ist `getconf` wegen POSIX `getconf PATH`
- Ein unbedarfter Nutzer sieht diese eingebauten Kommandos nicht

# Danke!

**URL:** http://cdrtools.sf.net/Files/

**Shell URL Sammlung:** http://schilytools.sf.net/bosh.html