Abstract

- The early days of Unix at Bell Labs and discussion of some of the key decisions made designing the shell language.
- Why we did what we did. What worked and what did not.
- Some of the shell innovations will be discussed as well as what we learned later.

Early days of Unix and design of sh

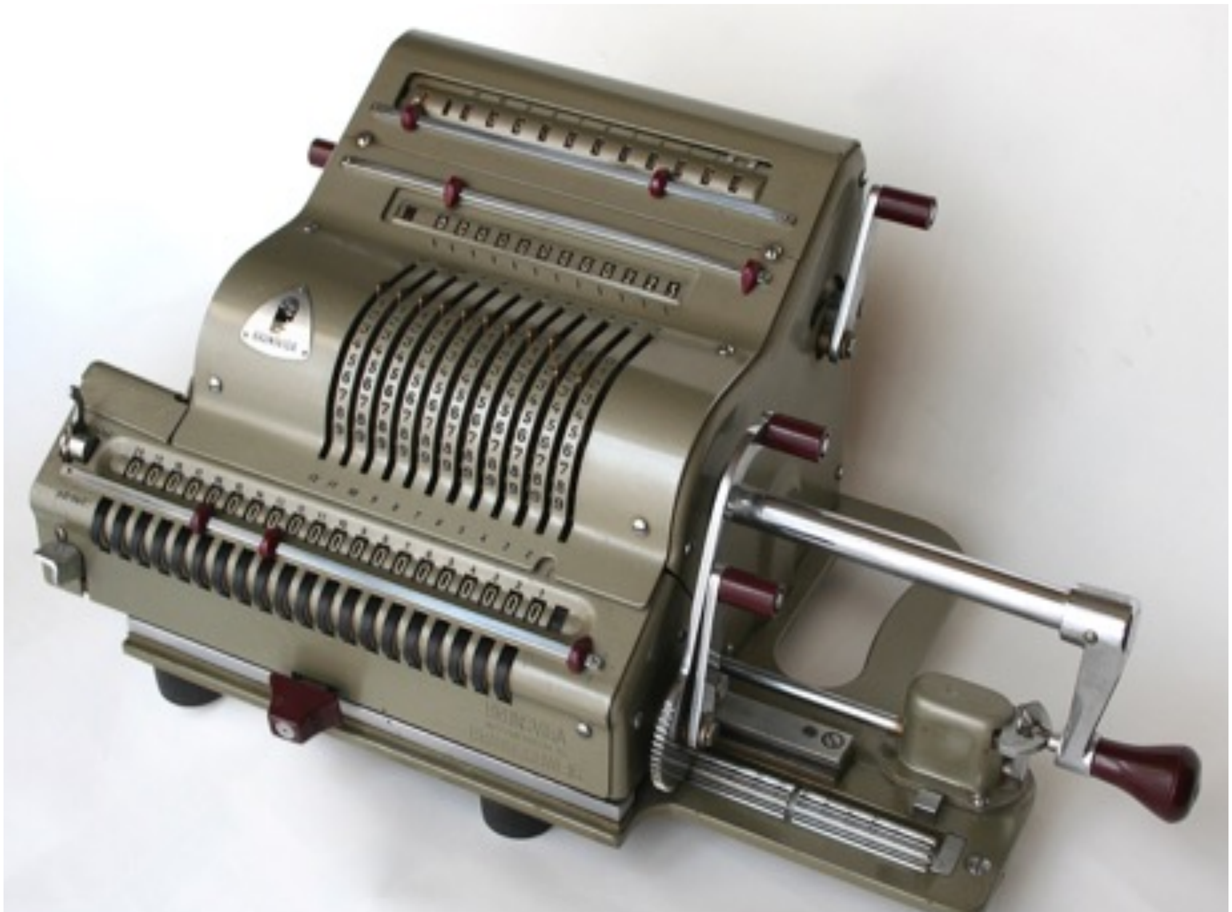
Stephen R. Bourne

Rally Ventures and ACM Queue EiC

NYC Bug

November 19th, 2015

Brunsviga at Mathematical Laboratory



How I got to Bell Labs

- At Cambridge with Wilkes and Wheeler
- Algebra systems for Lunar Theory (CAMAL)
- Z language and life game (Conway ‘70)
- Algol68C compiler and ZCODE
- Cambridge CAP operating system
- Arrived at Murray Hill January 1975

Unix Development

- Two adjoining rooms in 6th floor attic
- PDP 11/45
- Model 33 teletypes and Tektronix 4014
- uucp – no other network
- 300/1200/9600 baud modems at home
- Instant design and quality feedback
- Source code and man pages online
- Directory conventions (etc, bin, src, man, …)
- RP05 disks and vertical surfaces

Key Players

- Dennis Ritchie (C and drivers)
- Ken Thompson (kernel)
- Doug McIlroy (pipes)
- Joe Ossanna (nroff, troff)
- Brian Kernighan (awk, C book)
- Mike Lesk (tbl, lex, uucp)
- Steve Johnson (portable C, yacc)
- Al Aho (awk, yacc)
- Stu Feldman (F77, make)
- Peter Weinberger (awk)

Sixth edition 1975

- Written in C
- 40 system calls
- grep man page was 20 lines
- Used outside of Unix group
- ed, grep, sed
- cc *.c, interfaces as .h files
- db, cdb
- Simple shell

TABLE OF CONTENTS

Seventh edition and BSD

- 1976
  - Wrote sh and adb
- 1977
  - make, lint, awk, uucp
- 1978
  - 32 bit port and Berkeley
  - Seventh edition published
  - File system improvements
  - 32-bit port to Interdata
- 1977 – 1980
  - Bill Joy and the C shell
  - BSD virtual memory
  - BSD 3.0 and overnight install

Seventh edition release management

- Utility owned by last person who touched it
- Makefiles for everyone
- Some attempts to de-lint
- man pages, src, executables, libraries
- Directory structure
- But no source control or versioning
- Ran compiles from release tree

Original sh

- Thompson Shell
  - Script recording with simple GOTO
  - /etc/glob for wild card characters
  - Shell scripts are not filters
  - Script was standard input
  - No control flow or variables ($1 - $9, $a - $z)
- The UNIX Time-sharing System - A Retrospective by Dennis Ritchie

Why we started over and re-wrote sh

- Meeting in December in Murray Hill with Dennis
- Ken Thompson was in Berkeley for a year
- PWB Mashey shell started mid 1975
- All were patched versions of Ken’s sh
- Started writing code toward end of 1975
- First versions deployed in early 1976

sh as a language

- Typeless (like BCPL)
- Strings are first class and only citizens
- Delimited by blanks and reserved characters
- Serves as interactive and scripting language
- Provides programmable interface to the Unix system
  - Variables and substitution
  - Control flow
  - Signal management
  - Process management
  - eval

Shell Design and implementation

- ALGOL 68 concepts
  - Program flow
  - Closed forms (if … fi, case… esac)
  - Complete substitutions anywhere
- No limits on strings (or anything else)
- String quoting rules
- Return values used for conditionals (exit=)
- Performance and strings
- Memory allocation

Shell features

- Multi character variables
- Environment variables
- Here documents (<<) with EOF and substitution
- Scripts as filters
- Command substitution
- Commands as functions (later)

Memory management Unix at the time

- In 1975 we were running sixth edition Unix
- Did not use the C library
  - Mostly because I didn’t need to
  - Also because of malloc conflicts
- Considered yacc and lex but too heavyweight
- No stdio or string copy routines
- No setjmp and longjmp (added)

Memory management

- No string length or any other "arbitrary" length restrictions
- Overall flow
  - Read in element
  - Evaluate from internal tree recursively
  - Same from tty or file
- During evaluation anything goes
  - Language is recursive so C stack not helpful
  - Variable assignment of arbitrary length strings anywhere e.g.

```
X = ` cat <<!
        cd `pwd`
        for v do read v; echo $v; done
       !
`
```

- Interleaved heap and stack
- Stack used for permanent objects including parse tree and partially constructed strings

Memory Management credits

- A Partial tour through the UNIX shell
  - Geoff Collyer, University of Toronto
  - Suspect this based on AT&T system 3 shell
  - Based on bugs reported
- Annotated source by Akira Nakamura 1990

Memory allocation
(upside down)

sbrk from bottom of memory

sh heap storage at bottom of this area

stack for strings and parse tree

stack and heap are interleaved

C stack from top down

The hard bits

- Signals and process management

```
cat <<! &
xyz
!
```

- Quoting specification and exclamation
- Error recovery and reporting
- Debugging memory allocation
- Here documents
- Performance
- The usual corner cases

Quoting hell

- There are at least three conceptually distinct mechanisms
  - $ parameter substitution
  - argument splitting and parsing commands
  - file name generation (ls *)

- Suppose $1, $2 and $3 have the values
  - $1 = <>
  - $2 = <a b>
  - $3 = <*>
- Some choices are

arg	nsh	osh
$1	-	-
$2	<a><b>	<a><b>
$3	<...>	<...>
"$1"	<>	<$1>
"$2"	<a b>	<$2>
"$3"	<*>	<$3>
"\$1"	<$1>	<$1>
'$1'	<$1>	<$1>

Meta character rules

	\	$	`	’	"	*	/
'	n	n	n	t	n	n	n
`	y	n	t	n	n	n	n
"	y	y	y	n	t	n	n

Unix group conversion

- printing exit= after each command
- od (octal dump) hence “done”
- goto gone

  “The absence of a goto is going to be mourned loudly by many at BIS. We have a lot of COBOL and Assembler types here who don't seem able to live without it.” (1977 comment)
- wait command interruptable

  “An interruptable wait has been long awaited and welcomed. Maybe it should have a flag argument to make it uninterruptable if desired.” (1977 comment)
- sh scripts as filters (what it used to be)
- 32 bit porting 1978 and *0
- sbrk and more porting grief (things you don’t think of when you write the code)
  - asked Dennis what sbrk recovery was in fault routine
  - never well documented

Language battles - C vs sh

From srb Sat Mar 5 13:17:31 1977

More seriously wrt do … done

Since there seems to be no hope of C becoming an expression language (in which case it would be potentially more suitable for the shell) there is equally no hope that the shell and C will be the same language.

The worst problem is the if...else problem which in C requires a lookahead. The shell cannot afford to do lookahead since it is an interactive language.

‘word ‘ – ‘drow’ dangling ‘else’ and keeping if … fi

- Ways of dealing with the word drow “problem”

case a in ... esac	case a { ... }
for i do ... od	for i { ... }
for i in a b c do ... od	for i in a b c { … }
while c do … od	while c { … }

- Separating for and while (originally one construction) is reasonable in the command language context.

- Its use in programming languages is to look for a member of a set with some specific property; I cannot think of a use for this in the shell.

‘word ‘ – ‘drow’ dangling ‘else’ and keeping if … fi

For all of the above both forms can easily be provided.

```
if c                                   if c
then d                                           d
else e                                 else e
fi
```

Is the dangling else is better than the fi. One way out of dangling ‘else’ is to say that there are two constructions one with no else part and the other with a ‘then’ part

```
if c
          d

if c
then d
else e
```

‘then’ signals the coming of the else (so to speak). The dangling else has gone away but one has to remember that ‘then’ is written if there is an else part.

Keeping if … then … else

It seems rather radical to remove both the `structured programming' primitives (if, while). They are both used at present. The alternative for ‘if’ using ‘select’ is somewhat cumbersome and unintuitive.

Using && and || produces even more amazing looking programs. e.g.

```
if a; then b; else c; fi
```

Becomes

```
{{{a&&{b;set $rc=$r; true}}||c; set rc=$r;}; test rc = 0}
```

Keeping case … esac

The choice of words (select, switch, case, ...) seems unimportant although of these ‘case’ is shortest.

The proposed construction is more error prone and is likely to give surprising results. e.g.

```
case $1 { ... }
```

will distribute the case inside the {...} so that leaving out the ; is a disaster.

Also how does

```
case $1; {-x
          -y
          echo x or y
          }
```

parse? There is no natural way to extend to more than one case selector.

At present there is enough redundancy to find mistakes.

Associative memory aka Environment variables

- “The shell has had a keyword mechanism working from within the shell (the wrong place) for a couple of years.”
- “The reason for introducing environments was to make this mechanism uniform for shell procs and C programs.”
- Named variables and context
  - Debates over sh conventions
  - x=y cmd …
  - cc, make, and dd used x=y in its arguments
- You don’t have to know variables at each process level
- Unix process rules - no child surprises
- Keyword parameters should be settable either at the call of a cmd, or in an enclosing environment.

Environment variables

- A pcs should be able to have `local' names. i.e. names which are not passed on to children nor cause the behaviour of the child to be modified in any way.
- The names used by a child can be set by any ancestor of the child. It should not make any difference to the child how the name was set.
- It should be possible to set a name for use by descendants.
- A pcs should not have to be a postman for all the names which are being passed. It only need look at those intended for it. Others will automatically be passed on to children.

Environment variables – alternatives

Wed May 17 09:04:24 1978

Viewed from a C routine there are four levels of scope:

a) Automatic variables, whose life is the subroutines’;
b) External variables, whose life is the memory load;
c) Shell environment variables, whose life is the session;
d) Files, which last forever.

It would be nice if all of these looked the same. For example:

- a,b, and d are binary; c is string
- only c has "export" and "readonly”,
- a,b, and d are accessed by address;
- c has associative searches for "x=”.

We could have a storage class "session" which was like "extern” except it lives longer.

Unfortunately, this requires enormous system changes.

So make session variables look just like files, i.e. that a new set of routines sopen, sseek, sclose, sread, and swrite exactly image open, seek, ... except that they work on session files.

Environment variables
Fri May 12 01:44:05 EDT 1978

The shell has been modified to take advantage of the new UNIX environment passing stuff (notably execve).

As a consequence shell variables are initialized from the environment but are not transmitted back to the environment by default.

```
export name
```

causes name to be sent to the environment when processes are created

Early development - quoting

Thu Jun 3 18:21:29 EDT 1976

- In response to popular demand a number of changes have been made to the shell. Please let me know if there are any problems (srb).
- $ is no longer used as the quoting character; instead a \ is used. For example
  - echo \;   will print ;
  - echo \$ will print $
  - Within double quotes the meaning of \ is somewhat modified and the only characters escaped are " $ and newline.
- A new quoting mechanism '...' has been added that inhibits all interpretation of the enclosed string.   For example
  - echo '$1'   will print $1
  - echo '\\'   will print \\

Early development ….

- ‘here’ documents are processed in one of two ways depending on whether the string following << is quoted or not.
  - not quoted (e.g. <<!)
    - a \ is used to escape a newline, $ and the first character of the terminating string.
    - Parameter substitution occurs within the document.
  - quoted (e.g. <<\!) all characters are passed literally and no parameter substitution occurs.
- Command substitution is now written \`command\`
  - Nested uses now need to be quoted.

Early development ….

- Added built-in names $- $$ $# $! $?
- "$@" is the same as "$1" "$2" ...
- Error checking e.g.

  The notation ${...} is now checked more thoroughly. Also the default strings are only evaluated when needed so that e.g.

  ```
  echo ${d-`newfile`}
  ```

  only executes the command newfile if $d is not set.
- Interrupt handling e.g.
  - echo */*/* is now interruptable.
- Keyword parameters (x=y) only before cmd
- Performance improvements

  Fri Mar 4 17:02:57 EST 1977
  Some modifications have been made to the shell. One result is that it now runs up to twice as fast as the current shell. It also uses less space. As usual some bugs have gone and some more have appeared (although in both cases they escape my test programs)

The C language and libC

- Types
  - Structures like Algol68 not PL/1
  - Void
  - Pointers with explicit dereference
  - Casts
  - L and R values
  - Unions added
- Functions with local and global variables
  - No recursion
  - All functions returned a value
- Memory management
  - malloc, free
  - Strings done “by hand” no ’strcopy’
  - No array bound checking
- libC then stdio (later)
- Interfaces via .h files
  - Conventions for initializing global variables
  - Where instances are created
    - define vs declare

Various shells

- Original Thompson shell
  - 3 man pages
- sh
  - 6 man pages
- bash man pages and word count
  - 110 man pages
  - 4890   37094 329778
- POSIX and ksh
  - Designed by committee
  - Early 80s
- dash
  - A return to sanity ?

What I would have done differently and what worked

- Write code others can read (A68 macros)
- Memory management fragile when porting
- Functions as first class citizens
- Add some real support to debug scripts
- Using 8th bit of byte as quoted character marker

- Started somewhere and iterated
- Had real users and heard what they said
- Resist feature creep given design center
  - The shell can only solve so many problems
  - cat –v considered harmful (Pike)

Now what

- ACM Queue
  - Publishing for 10 years now
  - Focus is on problem not solution
  - Moving upstack
  - Publishing every other month via app
- Algol68C
  - Finish port
  - Incorporate libraries to attach the real world
  - Open source release

Thank you and Questions

Memory allocation

ß Base of heap at bottom of memory

Interleaved with stack items awaiting recovery

ß stakbsy # chain of stack blocks that have become
# covered up by heap allocation. `tdystak’
# will return them to the heap

ß Top of heap

ß stakbas # base of entire stack

ß stakbot # base of current stack item

ß staktop # top of stack

ß brkend # top of entire stack

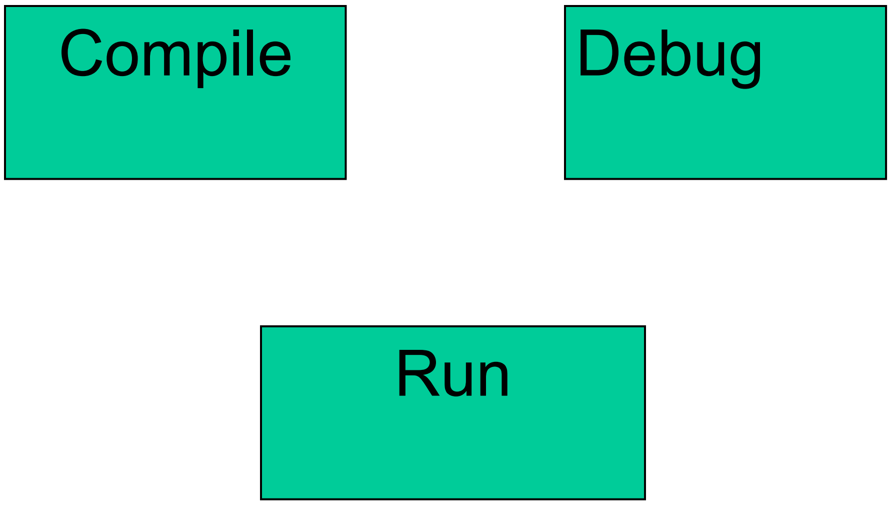
ß C stack from top of memory down

Stack and heap

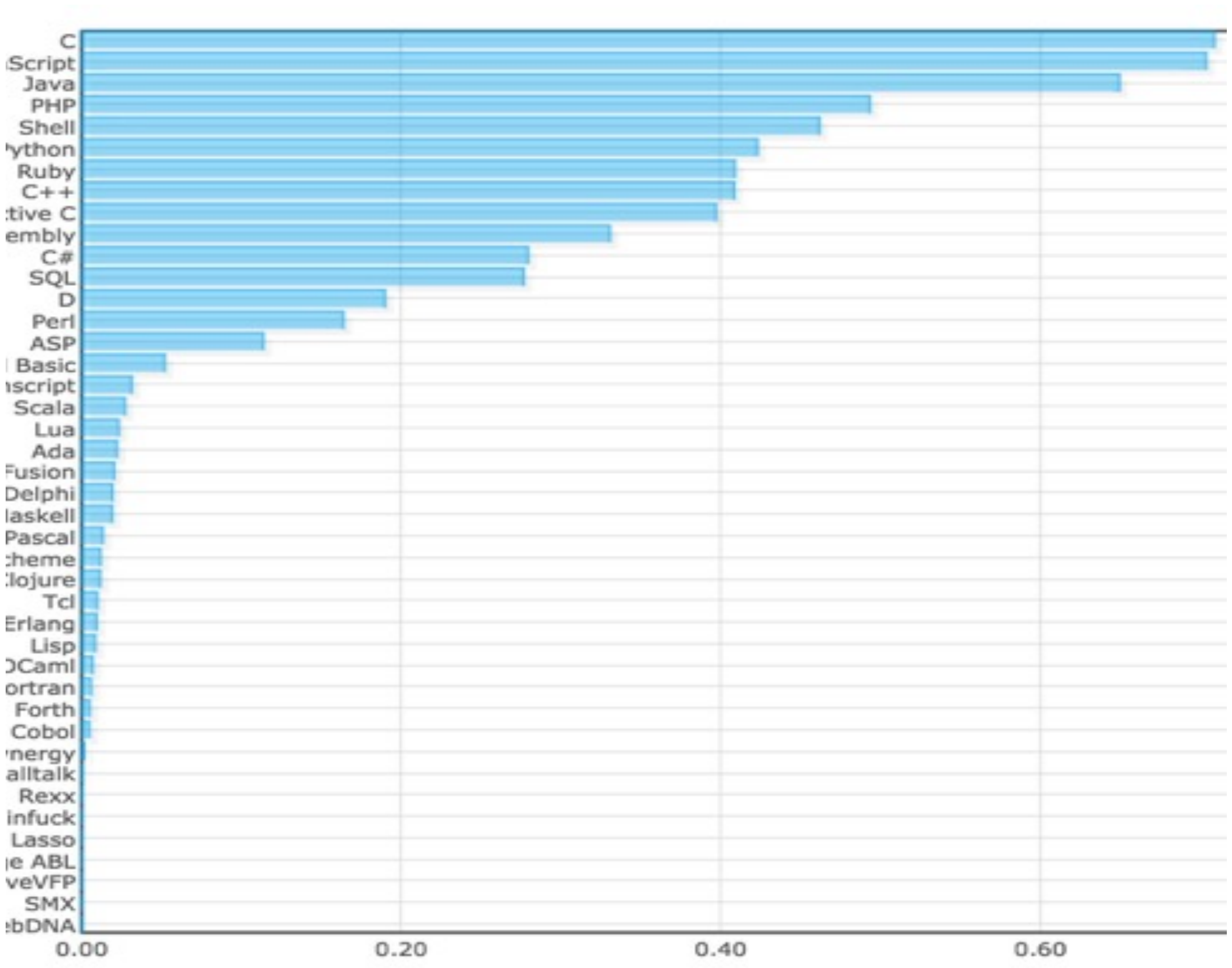
```
/*
 *          UNIX shell
 *
 *          S. R. Bourne
 *          Bell Telephone Laboratories
 */

/* To use stack as temporary workspace across
 * possible storage allocation (e.g. name lookup)

 * a)       get ptr from ‘relstak’
 * b)       can now use ‘pushstak’
 * c)       then reset with ‘setstak’
 * d)       ‘absstak' gives real address if needed
 */

#define          relstak()          (staktop-stakbot)
#define          absstak(x)         (stakbot+Rcheat(x))
#define          setstak(x)         (staktop=absstak(x))
#define          pushstak(c)        (*staktop++=(c))
#define          zerostak()         (*staktop=0)
```

adb – a debugger

- Written to debug Algol68C port
- Added overlays via exec
- adb could run on one machine debug another and be compiled on a third
- How to separate out representations



Usage by langpop



Complete list from 1977

- dynamic storage management (for variables &c.)
- control structures (at tty as well)
- parameter substitution + - ? =
- case pattern matching
- path searching
- built in file name matching (/etc/glob)
- no re-evaluation after $ substitution
- more general patterns (/sys/s?/...)
- general trap and fault handling
- control over child signals
- piping into or out of loops
- mail notification
- argument substitution eg nohup(a;b)
- efficient for loop and reads from files
- error recovery from sub command failure assured
- shell error handling can be controlled
- checks for cat >x >x and cat >x | wc
- cannot execute/cannot find distinguished
- general redirection e.g. 2>…
- input and execution traces; also no execution mode
- wait is interruptible
- cat >x hangs on open can be INTR’d