

# Some Classic UNIX Utilities With Some Life Left In Them

Ted Faber  
USC/ISI  
23 Oct 2003

# What Are We Talking About

---

Old School Stuff

Text-driven UNIX utilities

Tools To Work With

- parsing files
- arithmetic manipulations

# What's To Learn Here?

---

Traditional UNIX Design

## Simple filters

- simple tools combine in versatile ways
- tools do one things well

## Little Languages

- small, tailored syntax for specific tasks
- filters can combine these, e.g. groff

## Not so little languages

# Outline

---

IBM standard

## File parsing tools

- 3 tool comparison

## Calculation languages

- 3 tool survey

# File Processing

---

pipes, perl, and lex

Task - split out 3 fields from colon-delimited file

- e.g. /etc/passwd

Contrast 3 approaches

- simple filters:
  - cut, sed, sort, grep
- perl
- lex

# Filters

---

The Lifeblood of UNIX

Simple programs with a piped, plain text interface

Do one thing well

UNIX pipes allow composition to do complex things

E.g., a word counter:

```
■ tr '\t' '\n' < file | tr 'A-Z' 'a-z' | sort | uniq -c | sort -r -n +0 | head -10
```

# Perl

---

The Swiss Army Chainsaw

Lots of common idioms under one roof

- awk associative arrays
- sed regular expression manipulations
- system call interface
- shell escapes

Structured to be intuitive to people who use those programs

# Lex

---

The Old Chainsaw

Language to write regular expression parsers

Output language is C (or C++)

Enormously simplifies parser generation

- usually faster than perl
- significantly harder to debug

# Simple Solutions

---

Just Parsing

cut

- `cut -d : -f 1,5,6 file`

perl

- `perl -F: -nae 'print(join(":", @F[0,4,5]), "\n");' file`

lex (hang on...)

# Lex Solution

---

Just Parsing

```
% {  
int i = 0;  
% }  
%%  
\n { printf("\n"); i = 0; }  
[^:\n]+ {  
    if ( i == 0 || i == 4 || i == 5)  
        printf(((i != 5) ? "%s:" : "%s"), yytext);  
    i++;  
}  
::;  
%%
```

---

```
$ make passwd1  
lex -t passwd1.1 > passwd1.c  
cc -O -pipe  passwd1.c -ll -o passwd1
```

# What If The Problem Changes?

---

Times they are...

Let's add:

- Field cleanup - name has ',','s
- Output sorting
- Comments - with a leading '#'

Filters:

- `grep -v '^#' file | cut -d : -f 1,5,6 | sed 's/, [^:]*//g' | sort`

# Perl Solution

---

---

Parsing, Cleanup, Sorting

```
#!/usr/bin/perl

while (<>) {
    next if /^#/;
    chomp;
    s/,[^:]*//g;
    @F=split(':', $_);
    push(@lines, join(':', @F[0,4,5]));
}

for (sort @lines) { print "$_\n"; }
```

# Lex???

---

## Parsing, Cleanup, Sorting

```
% {
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#define BSIZE 125
int i = 0;
char buf[BSIZE];
int off = 0;
char **lines = 0;
int lcap = 0;
int nlines = 0;
% }
%%
^#.*\n ;
\n {
    if ( nlines >= lcap ) {
        lcap += 25;
        lines = realloc(lines, lcap * sizeof(char *));
    }
    if ( !lines ) {
        perror("out of memory!");
        exit(20);
    }
    if (!(lines[nlines++] = strdup(buf))) {
        perror("strdup failed?!");
        exit(20);
    }
    off = i = 0;
}
[^\n]+ {
    if ( i == 0 || i == 4 || i == 5 )
        off += sprintf(buf+off, BSIZE-off, ((i!=5) ? "%s:" : "%s"),
            yytext);
    i++;
}
: ;
%%

int sc(const void *a, const void *b) {
    const char *const *aa = a;
    const char *const *bb = b;

    return strcmp(*aa,*bb);
}

int main(int argc, char ** argv) {
    int i;

    while ( yylex() )
        ;
    qsort(lines, nlines, sizeof(char *), sc);
    for ( i = 0; i < nlines; i++ )
        printf("%s\n", lines[i]);
}
```

# Lex & C++

---

## Parsing, Cleanup, Sorting

```
% {
#include <iostream>
#include <string>
#include <set>
string s;
set<string> lines;
int i = 0;
% }
%%
^#.*\n ;
\n { lines.insert(s); s = ""; i = 0;}
[^:\n]+ {
    if ( i == 0 || i == 4 || i == 5 ) {
        s += yytext;
        string::size_type pos = s.find(',');
        if ( pos != string::npos ) s.erase(pos);
        if ( i != 5 ) s += ":";
    }
    i++;
}
::;
%%
int main(int argc, char ** argv) {
    while ( yylex() )
        ;
    copy(lines.begin(), lines.end(), ostream_iterator<string>(cout, "\n"));
}
```

# Scorecard

---

can't tell the players

## Filters

- solve a lot of problems
- communication language very simple
- N. B. No model change for our example problem

## Perl

- some ready-made glue language for common filters
- intuitive for filter users
- can constrain solutions

## Lex

- rich output language (C/C++)
- less data abstraction means more hands on work

## Mix and Match

- perl | sort

# Numbers

---

in the days before spreadsheets...

dc

- RPN desk calculator

bc

- full-featured calculation language

nickle

- calculator language gone mad

# dc - desk calculator

---

the thing you wind up in when you mistype cd

## RPN calculator

- 2 2 + leaves 4 on the stack
- HP and Forth guys are happy

## High precision fixed point

- precision is user configurable
- defaults to integers

## Arbitrary bases

## Simple arcane macro language

# Average Calculation in dc

---

average the stack

```
0st0sn  
[lt+stln1+snz0<az0=d]sa  
[ltln/p0st0sn]sd
```

---

```
$ dc dc.avg -  
1 2 3 4 5 6 7 8 9 10 lax  
5  
c 5 k  
1 2 3 4 5 6 7 8 9 10 lax  
5.50000
```

# bc - a real calculator language

---

tougher to mistype, easier to use

Infix, C-style syntax

Variables, Arrays and Functions

Decimal, Fixed Point Arithmetic

Reasonable Numeric Scripting Language

# Average Calculation in bc

---

read and average a list of numbers

```
define avg (n, x[]) {
    auto i, t;
    t = 0;
    if ( n < 1 ) return;
    for (i=0;i<n; i++) t += x[i];
    return t/n;
}
define read_avg() {
    auto n, x, a[];
    n=0;
    while ( x = read() ) a[n++] = x;
    print "avg is: ", avg(n, a[]), "\n";
}
```

---

```
$ bc bc.avg
read_avg()
[ input 1..10]
avg is: 5
scale=10
read_avg()
[ input 1..10]
avg is: 5.5000000000
```

# Nickle

---

calculators gone mad

Infix, C-style syntax

Rich numeric model:

- integers
- rationals
- arbitrary exponent precision reals

Programming Language Features

- data structures: structs, arrays, slices, unions
- database support
- functions
- libraries
- threads

<http://www.nickle.org>

# Average Calculation in nickle

---

averaging and some data types

```
rational function avg(rational[*] a) {  
    rational t = 0;  
    int i;  
    int n = dim(a);  
  
    if ( n > 0 ) for (i=0; i< n; i++) t+= a[i];  
    else return 0;  
    return t/n;  
}
```

---

```
$ nickle -f nickle.avg  
> rational[*] q = { 1,2,3,4,5,6,7,8,9,10}  
[10] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}  
> avg(q)  
5.5  
> printf("%v\n", avg(q))  
(11/2)
```

---

```
$ nickle  
> rational y = 1/3  
0.{3}  
> printf("%v\n", y)  
(1/3)  
> real z = imprecise(1/3)  
0.3333333333333333
```

# Calculator Scorecard

---

numbers wrapup

dc

- very simple interface
- good accuracy

bc

- more rich language
- better scripting

nickle

- very rich language
- suitable for real numeric applications

# Conclusions

---

wrap-up

## Power of Modularity

- filters
- program compilers

## Languages at Many Levels

- RPN
- regular expressions
- C/C++
- XML