Random thoughts on scripting languages

Brian Kernighan
Department of Computer Science
Princeton University

Revisionist history of programming languages

- 1940's machine language
- 1950's assembly language
- 1960's high-level languages: Algol, Fortran, Cobol, Basic
- 1970's systems programming: *C*
- 1980's object-oriented: *C++*
- 1990's strongly-hyped: Java
- 2000's copycat languages: *C#*
- 2010's ???

Scripting languages

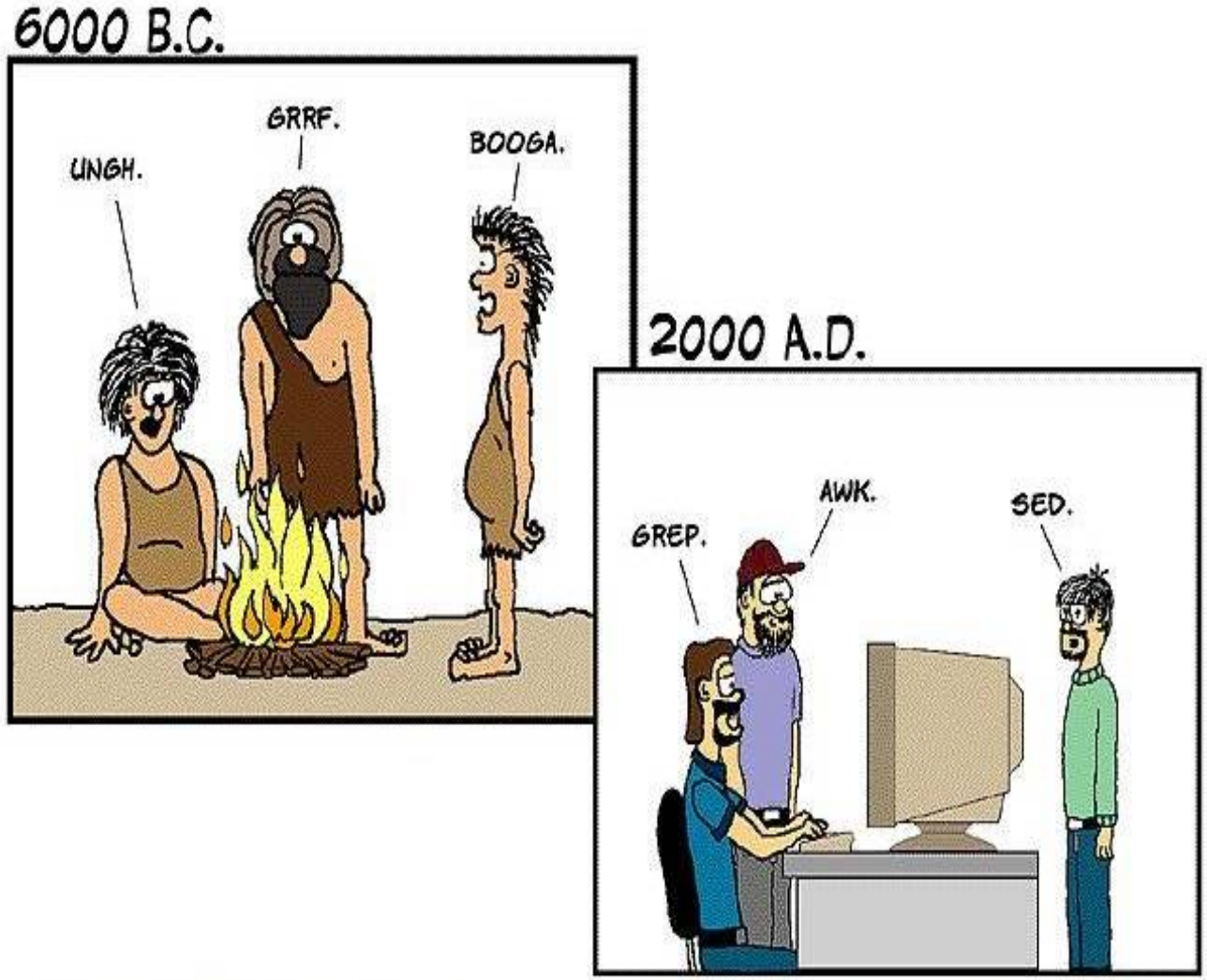
- **"… scripting is a lot like obscenity. I can't define it, but I'll know it when I see it."**
  - Larry Wall, creator of Perl

- **characteristics**
  - text as a basic (or only) data type
  - regular expression support, perhaps even in syntax
  - associative arrays as a basic aggregate type
  - minimal use of types, declarations, and other baggage
  - usually interpreted instead of compiled
- **examples**
  - shell
  - Awk
  - Perl, PHP, Ruby
  - Tcl
  - Python
  - Visual Basic, (VB|W|C)Script, PowerShell
  - Javascript

LISP, Scheme, functional languages

this page intentionally left blank

EVOLUTION OF LANGUAGE THROUGH THE AGES.



HTTP://WWW.USERFRIENDLY.ORG/

AWK – the first (or second?) scripting language

Al Aho, Brian Kernighan, Peter Weinberger (Bell Labs, 1977)

- **intended for simple data processing:**

  selection, validation:

  "Print all lines longer than 80 characters"

  ```
  length > 80
  ```

  transforming, rearranging:

  "Replace the 2nd field by its logarithm"

  ```
  { $2 = log($2); print }
  ```

  report generation:

  "Add up the numbers in the first field, then print the sum and average"

  ```
  { sum += $1 }
  END  { print sum, sum/NR }
  ```

A typical problem

- data: thousands of lines like

```
8/27/1883     Krakatoa         8.8
5/18/1980     Mt St Helens     7.6
3/13/2009     Costa Rica       5.1
```

- task: Find all the earthquakes with magnitude greater than 6
- how do you proceed?
- do it by hand
- write a program in [insert favorite programming language here]
- use Awk

```
awk  '$3 > 6'  < inputfile
```

"A programming language that doesn't change the way you think is not worth learning."

Alan Perlis, *Epigrams on Programming*

Structure of an AWK program

- a program is a sequence of pattern-action statements

```
pattern   { action }
pattern   { action }
…
```

- a pattern is a regular expression, numeric expression, string expression or combination
- an action is executable code, similar to C
- usage:

```
awk 'program' [ file1 file2 ... ]
awk -f progfile [ file1 file2 ... ]
```

- operation:

```
for each file
  for each input line
    for each pattern
      if pattern matches input line
        do the action
```

AWK features

- **input is read automatically**
  - across multiple files
  - lines split into fields $1, ..., $NF; **$0 for whole line**
- **variables contain string or numeric values**
  - no declarations
  - type determined by context and use
  - initialized to 0 and empty string
  - built-in variables for frequently-used values (e.g., NF, NR)
- **operators similar to C, but work on strings or numbers**
  - coerce type according to context
- **control flow statements similar to C**
- **associative arrays (arbitrary subscripts)**
- **regular expressions like egrep**
- **built-in and user-defined functions**
  - arithmetic, string, regular expression, text edit, ...
- **printf for formatted output**
- **getline for input from files or processes**

Basic AWK programs

```
{ print NR, $0 }          precede each line by line number
{ temp = $1; $1 = $2; $2 = temp; print }   flip $1, $2
{ print $NF }             print last field

NF > 4                    print if more than 4 fields
$NF > 4                   print if last field greater than 4
/regexpr/                 print matching lines (egrep)
$1 ~ /regexpr/            print lines where first field matches
NF > 0 {print $1, $2}     print two fields of non-empty lines

END { print NR }          line count

      { nc += length($0) + 1; nw += NF }     wc command
END { print NR, "lines", nw, "words", nc, "characters" }

length($0) > max { max = length($0); line = $0 }
END      { print max, line }          print longest line
```

Associative arrays (=hash, dictionary, hashtable)

- array subscripts can have <u>any</u> value, not just integers

Input:

```
pizza       200
coke        50
beer        100
beer        150
pizza       500
beer        50
```

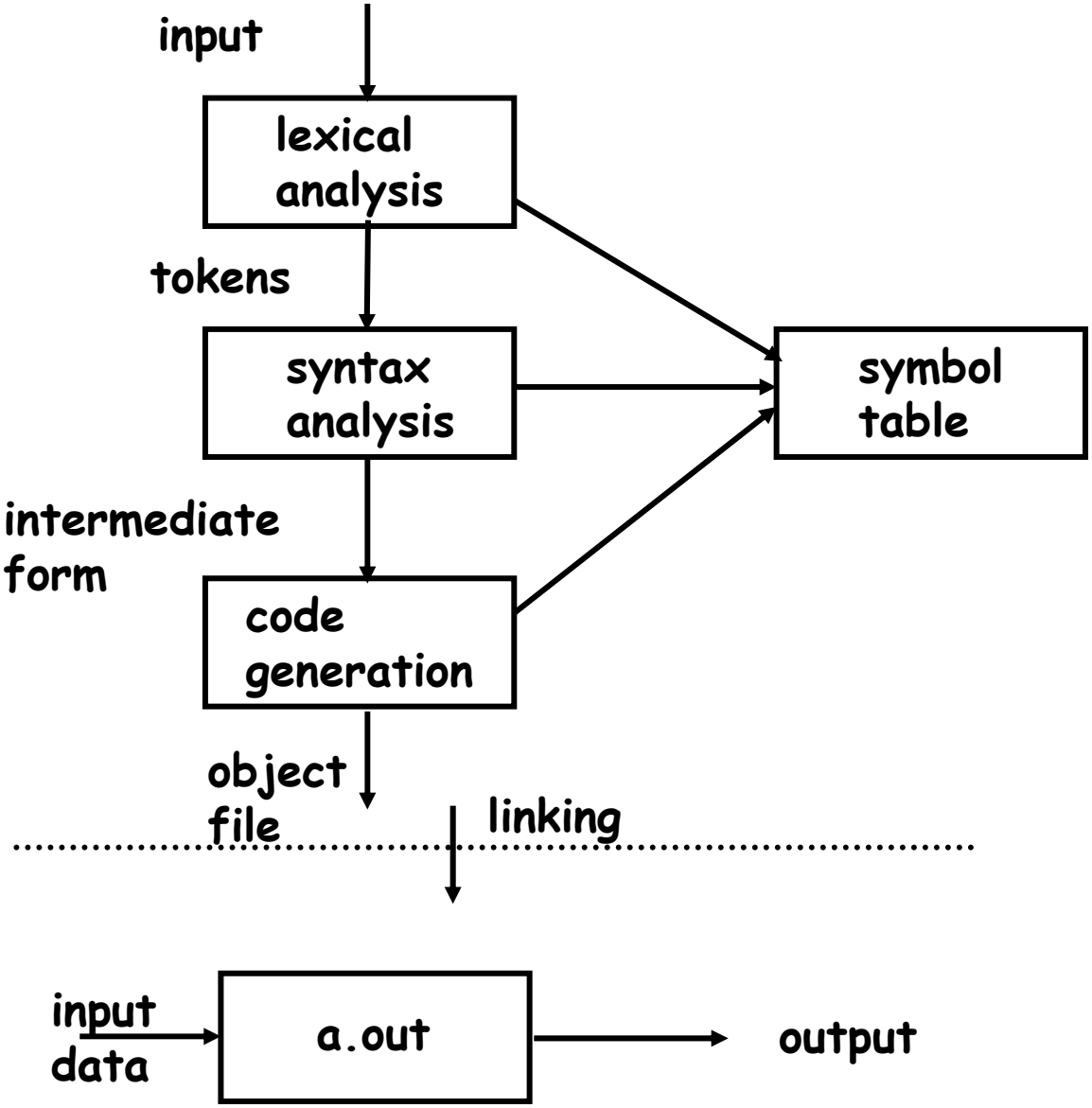
Program:

```
        { amount[$1] += $2 }
    END { for (name in amount)
              print name, amount[name] }
```

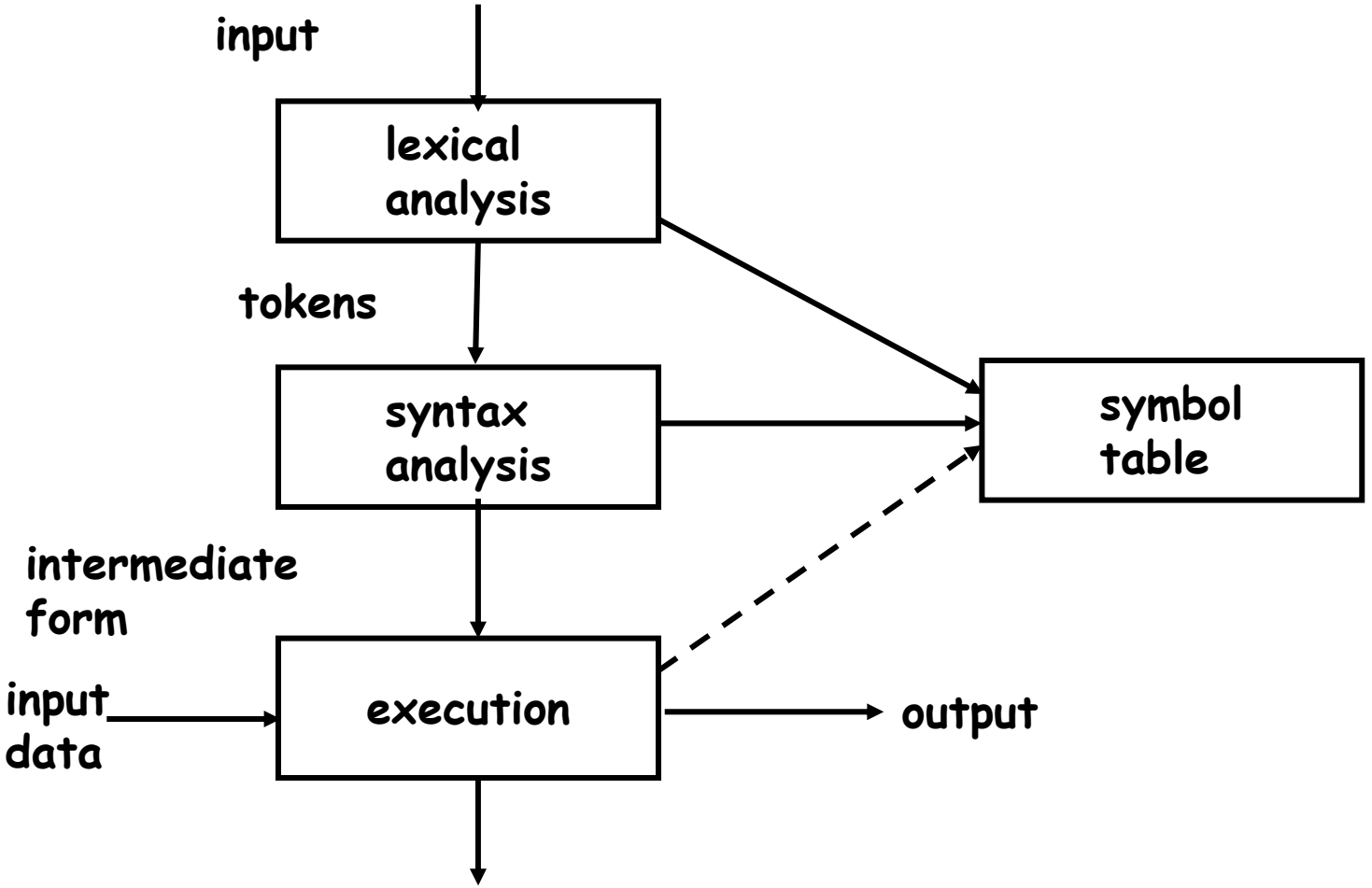
Output:

```
beer        300
pizza       700
coke        50
```

Anatomy of a compiler



Anatomy of an interpreter



Implementation notes

- **C + YACC (~6300 lines total)**
  - used to have a LEX lexical analyzer, now hard-coded C
- **parsing creates a parse tree**
- **interpreter walks the tree**
  - at each node, interpret children (recursion),
  - do operation of node itself, return result to caller
- **alternatively, could**
  - run directly from the program (shell, TCL)
  - generate byte code output to run separately (Java)
  - generate byte code (Python?)
  - generate C, C++, ...
- **compiled code runs faster**
  - but compilation takes longer, needs object files, less portable, …
- **interpreters start faster, but run slower**
  - for 1- or 2-line programs, an interpreter is better
  - on the fly / just in time compilers merge these

Testing: making sure it works

- **~ 1000 tests, run with a single command**

- **regression tests**
  - does it work the same as the previous version does?
- **known output tests**
  - does it produce the same output as an independent mechanism?
- **bug fix tests**
  - a test for each bug and new feature
- **stress tests**
  - does it handle perverse, huge, illegal, random cases?
- **coverage tests**
  - are all statements executed by some test?
- **performance tests**
  - does it run at the same speed or better?

Advice for testing

- mechanize
- make test output self-identifying
- make sure you can reproduce a test that fails
- add a test for each bug
- add tests for each new feature or change
- never throw away a test
- make sure that your tester reports progress
- make your tests portable
- check your tests and scaffolding often
- keep records

Using Awk for testing RE code in Awk

- **regular expression tests are specified in a specialized language:**

```
^a.$     ~       ax
                 aa
         !~      xa
                 aaa
                 axy
```

- **each test is converted into a command that exercises Awk:**

```
echo 'ax' | awk '!/^a.$'/ { print "bad" }'
```

- **illustrates**
  - mechanization
  - little languages
  - programs that write programs

Record of bug fixes

- a record of all bug fixes since August 1987

Nov 22, 2003:

- fixed a bug in regular expressions that dates (so help me) from 1977; it's been there from the beginning. an anchored longest match that was longer than the number of states triggered a failure to initialize the machine properly. many thanks to moinak ghosh for not only finding this one but for providing a fix, in some of the most mysterious code known to man.
- fixed a storage leak in call() that appears to have been there since 1983 or so -- a function without an explicit return that assigns a string to a parameter leaked a Cell. thanks to moinak ghosh for spotting this very subtle one.
- ...

Feb 21, 2007:

- fixed a bug in matching the null RE in sub and gsub. thanks to al aho who actually did the fix (in b.c), and to wolfgang seeberg for finding it and providing a very compact test case.
- fixed quotation in b.c; thanks to Hal Pratt and the Princeton Dante Project.

An important bug fix...

```
/* lasciate ogne speranza, voi ch'intrate. */

p = f->posns[s];
for (i = 1; i <= *p; i++) {
  if ((k = f->re[p[i]].ltype) != FINAL) {
    if ((k == CHAR && c == ptoi(f->re[p[i]].lval.np))
     || (k == DOT && c != 0 && c != HAT)
     || (k == ALL && c != 0)
     || (k == EMPTYRE && c != 0)
     || (k == CCL && member(c,(char *)f->re[p[i]].lval.up))
     || (k == NCCL && !member(c,(char*)f->re[p[i]].lval.up) && c!=0 && c!=HAT)) {
      q = f->re[p[i]].lfollow;
      for (j = 1; j <= *q; j++) {
        if (q[j] >= maxsetvec) {
          maxsetvec *= 4;
          setvec = (int *) realloc(setvec, maxsetvec * sizeof(int));
          tmpset = (int *) realloc(tmpset, maxsetvec * sizeof(int));
          if (setvec == 0 || tmpset == 0)
            overflo("cgoto overflow");
        }
        if (setvec[q[j]] == 0) {
          setcnt++;
          setvec[q[j]] = 1;
        }
      }
    }
```

Text formatting

**Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Vestibulum volutpat nisi at neque.
Proin molestie metus quis libero.
Nam consectetur pretium nisi.
Nullam consequat pulvinar est.**

**Vivamus et neque dapibus tortor pretium dictum.
Integer dictum neque sed nisi.
Integer pellentesque lorem id mauris.**

**Nam posuere congue turpis.
Pellentesque placerat lectus ac magna.
Praesent posuere ultricies tortor.
Etiam nec eros at ipsum pretium commodo.**

**Duis sodales diam in mi.
Integer iaculis nulla et arcu.**

**Morbi sodales nulla vel justo.
Fusce tincidunt tellus non erat.
Aliquam rutrum ipsum id quam.
Fusce placerat arcu quis nunc.
Ut imperdiet dictum est.**

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum volutpat nisi at neque. Proin molestie metus quis libero. Nam consectetur pretium nisi. Nullam consequat pulvinar est.

Vivamus et neque dapibus tortor pretium dictum. Integer dictum neque sed nisi. Integer pellentesque lorem id mauris.

Nam posuere congue turpis. Pellentesque placerat lectus ac magna. Praesent posuere ultricies tortor. Etiam nec eros at ipsum pretium commodo.

Duis sodales diam in mi. Integer iaculis nulla et arcu.

Morbi sodales nulla vel justo. Fusce tincidunt tellus non erat. Aliquam rutrum ipsum id quam. Fusce placerat arcu quis nunc. Ut imperdiet dictum est.

Awk text formatter

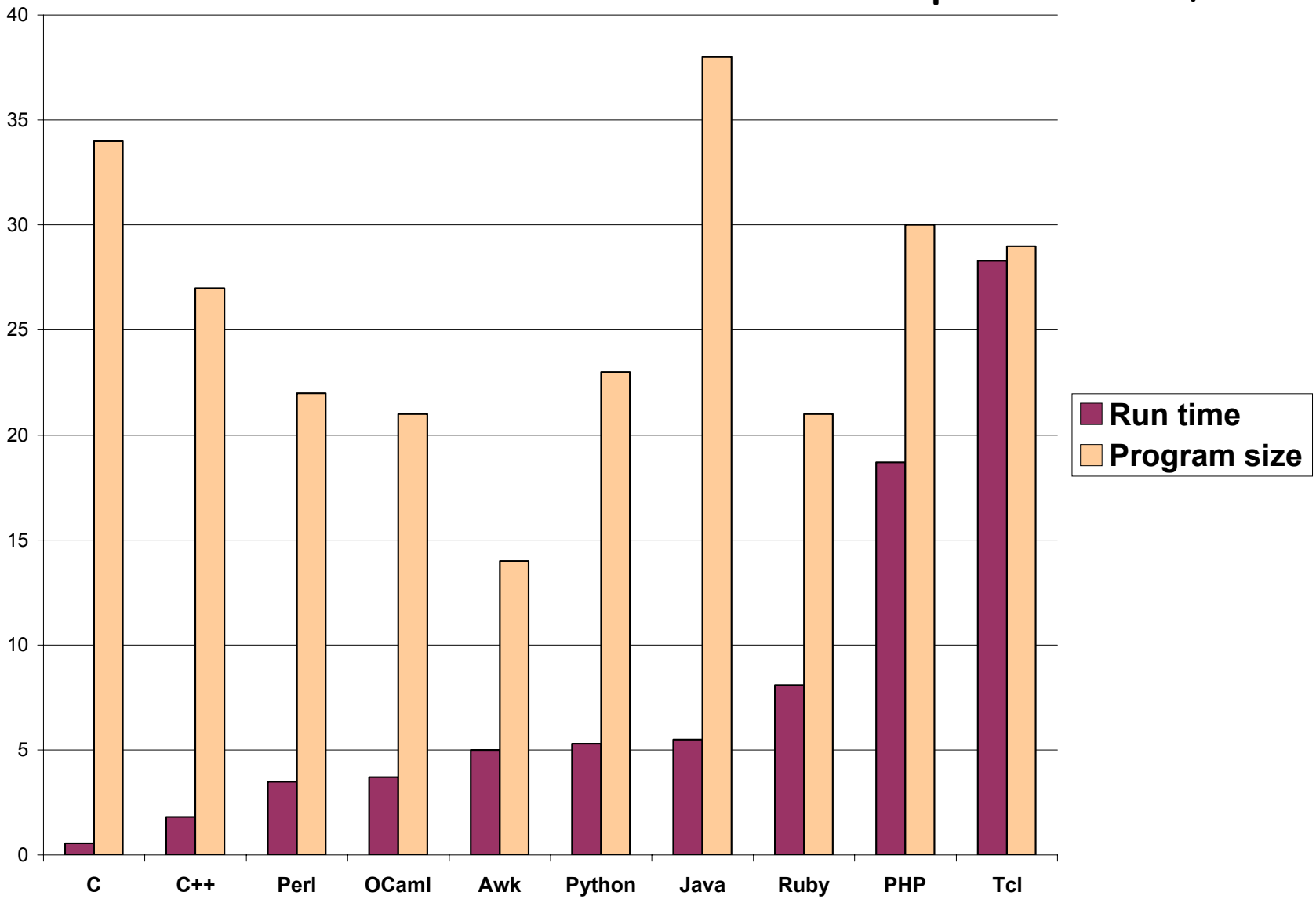
```
# format text into 60-char lines
/./  { for (i = 1; i <= NF; i++) addword($i) }
/^$/ { printline(); print "" }
END  { printline() }

function addword(w) {
    if (length(line) + length(w) > 60)
        printline()
    line = line space w
    space = " "
}

function printline() {
    if (length(line) > 0)
        print line
    line = space = ""
}
```

How fast does it run?

input: 31K lines, 4.5 MB



Lessons (1)

- **people use tools in unexpected, perverse ways**
  - compiler writing
  - implementing languages
  - object language
  - first programming language

- **the existence of a language encourages programs to generate it**
  - machine generated inputs stress a program differently than people do

Lessons (2)

- **mistakes are inevitable and hard to change**
  - function syntax
  - concatenation syntax
  - ambiguities, especially with >
  - difficulty of changing a "standard"
  - creeping featurism from user pressures

- **standardization is hard**
  - there is a POSIX standard for Awk
  - awk, gawk, mawk, tawk, busybox awk, ... all differ in places

- **internationalization is hard**

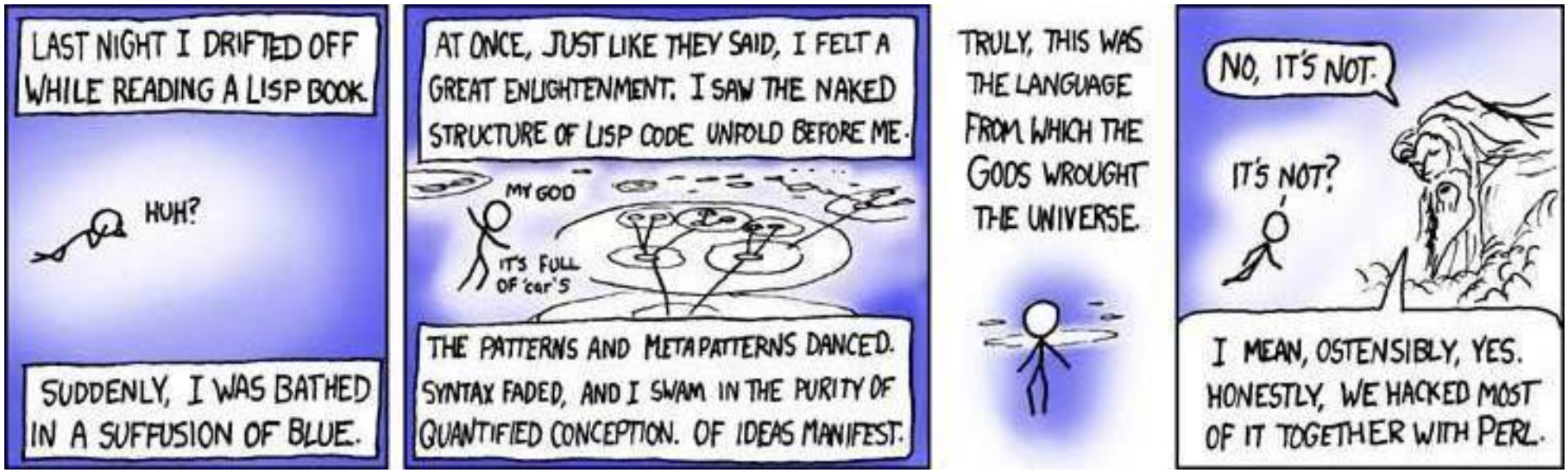
```
$1 ~ /[全-旅]/ { ... }
```

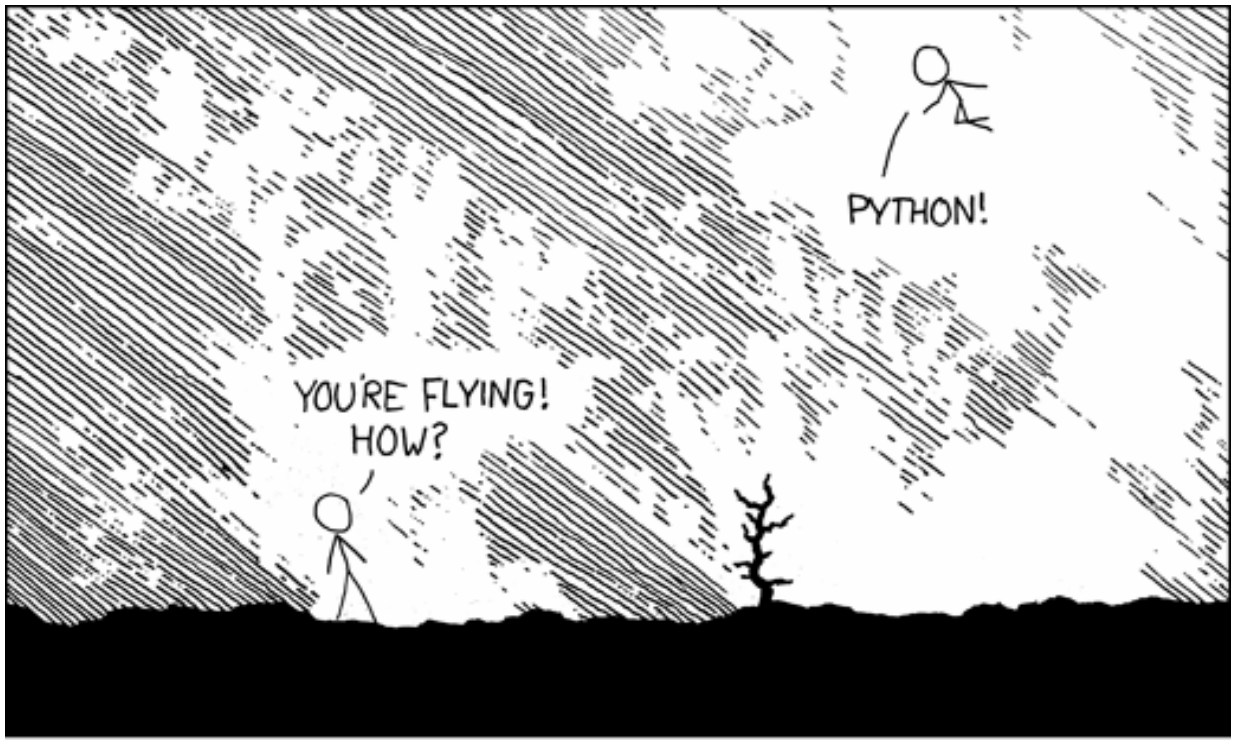
"One thing [the language designer] should not do is to include untried ideas of his own."

C. A. R. Hoare, *Hints on Programming Language Design*, 1973

Other scripting languages

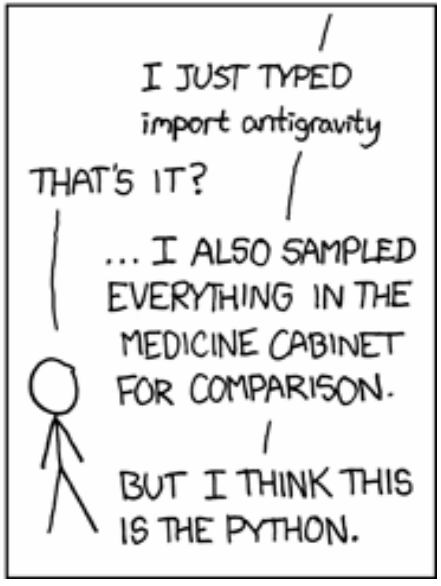
- **Perl**
  - in part a reaction to things missing from Awk
  - "Perl is Awk with skin cancer" (Henry Spencer)
- **Python**
  - "in Guido's mind was inspired positively by ABC, but in the Python community's mind was inspired negatively by Perl." (Larry Wall)
- **PHP**
  - in part a simplification of Perl
  - "takes the worse-is-better approach to dazzling new depths" (Larry Wall)
- **Ruby**
  - "it's just plain impossible to design a perfect language"
    (Yukihiro Matsumoto)
- **Javascript**
  - in part a reaction to Java for applets?
  - "Makes Javascript suck less" (marketing slogan for MochiKit)


LAST NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.
HUH?
SUDDENLY, I WAS BATHED IN A SUFFUSION OF BLUE.
AT ONCE, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE UNFOLD BEFORE ME.
MY GOD
IT'S FULL OF 'car'S
THE PATTERNS AND META PATTERNS DANCED. SYNTAX FADED, AND I SWAM IN THE PURITY OF QUANTIFIED CONCEPTION. OF IDEAS MANIFEST.
TRULY, THIS WAS THE LANGUAGE FROM WHICH THE GODS WROUGHT THE UNIVERSE.
NO, IT'S NOT.
IT'S NOT?
I MEAN, OSTENSIBLY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.


PYTHON!
YOU'RE FLYING! HOW?


I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"


I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?


I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

The future of scripting languages

- **they will continue to grow in importance**
- **they will continue to grow in number**
  - though only a handful will be widely used
- **everyone should know a few**
- **it probably doesn't matter which ones**
- **my current choices**
  - Python for broad utility, ease of use, comprehensible code
  - Javascript for client-side web programming
  - Awk for the most bang for the buck

Good reading

- **Scripting: Higher level programming for the 21st century**
  - John Ousterhout, IEEE Computer, March 1998

- **Programming is hard, let's go scripting**
  - Larry Wall, perl.com, December 2007

- **Hints on programming language design**
  - Tony Hoare, www.cs.berkeley.edu/~necula/cs263/handouts/hoarehints.pdf
    December 1973