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

Alan Perlis, *Epigrams in Programming*

Alan Perlis, 1922–1990  
*first head of CMU CS department*  
*first president of ACM*  
*first Turing Award winner*

AWK

- a language for pattern scanning and processing  
  - Al Aho, Brian Kernighan, Peter Weinberger, at 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"  
  ```bash
  { $2 = log($2); print }
  ```
- report generation:  
  "Add up the numbers in the first field,  
  then print the sum and average"  
  ```bash
  { sum += $1 }  
  END { print sum, sum/NR }
  ```
Structure of an AWK program:

- a sequence of pattern-action statements

  \[
  \text{pattern} \quad \{ \text{action} \} \\
  \text{pattern} \quad \{ \text{action} \} \\
  \ldots
  \]

- "pattern" is a regular expression, numeric expression, string expression or combination of these
- "action" is executable code, similar to C

usage:

\[
\text{awk} \ '\text{program}' \ [ \text{file1 file2 ...} ] \\
\text{awk} \ -f \ \text{progfile} \ [ \text{file1 file2 ...} ]
\]

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 are split into fields ($1, ..., $NF; $0 for whole line)
- variables contain string or numeric values (or both)
  - no declarations: type determined by context and use
  - initialized to 0 and empty string
  - built-in variables for frequently-used values
- operators work on strings or numbers
  - coerce type / value according to context
- associative arrays (arbitrary subscripts)
- regular expressions (like egrep)
- control flow statements similar to C: if-else, while, for, do
- 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, part 1

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

NF > 0  
print non-empty lines
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

Basic AWK programs, part 2

```
NF > 0 {print $1, $2}  
print two fields of non-empty lines
END { print NR }  
line count

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

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

- if-else, while, for, do...while, break, continue
  - as in C, but no switch

- for (i in array)
  - go through each subscript of an associative array

- next
  - start next iteration of main loop

- exit
  - leave main loop, go to END block

```awk
{ sum = 0
  for (i = 1; i <= NF; i++)
    sum += $i
  print sum
}
```

```awk
{ for (i = 1; i <= NF; i++)
  sum += $i
}
END { print sum }
```

Awk text formatter

```bash
#!/bin/sh
# f - format text into 60-char lines
awk '    
/./ { 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 = ""
}
' "$@"
```
Arrays

• common case: array subscripts are integers

• reverse a file:

```bash
{ x[NR] = $0 }    # put each line into array x
END { for (i = NR; i > 0; i--)
    print x[i] }
```

• make an array:

```bash
n = split(string, array, separator)
```
- splits "string" into array[1] ... array[n]
- returns number of elements
- optional "separator" can be any regular expression

Associative Arrays

• array subscripts can have any value, not just integers
• canonical example: adding up name-value pairs

• input:

- pizza    200
- beer     100
- pizza    500
- beer     50

• output:

- pizza    700
- beer     150

• program:

```bash
{ amount[$1] += $2 }
END { for (name in amount)
    print name, amount[name] | "sort +1 -nr"
}
```
Anatomy of a compiler

input

lexical analysis

tokens

syntax analysis

intermediate form

symbol table

object file

linking

input data

a.out

output

Anatomy of an interpreter

input

lexical analysis

tokens

syntax analysis

symbol table

intermediate form

execution

input data

output
YACC and LEX

- languages/tools for building [parts of] compilers and interpreters

- **YACC**: "yet another compiler compiler" (S. C. Johnson, ~ 1972)
  - converts a grammar and semantic actions into a parser for that grammar

- **LEX**: lexical analyzer generator  (M. E. Lesk, ~ 1974)
  - converts regular expressions for tokens into a lexical analyzer that recognizes those tokens

- parser calls lexer each time it needs another input token
- lexer returns a token and its lexical type

- when to think of using them:
  - real grammatical structures (e.g., recursively defined)
  - complicated lexical structures
  - rapid development time is important
  - language design might change

YACC-based calculator

```c
#include <stdio.h>
#include <ctype.h>
main() { yyparse() }
yyerror() { /* called for yacc syntax error */

#define YYSTYPE double /* data type of yacc stack */

%{

%token NUMBER
%left '+' '-' /* left associative, same precedence */
%left '*' '/' /* left associative, higher precedence */
%
list:  expr '
'         { printf("\t%.8g\n", $1); }   
| list expr '
'    { printf("\t%.8g\n", $2); } 
|       ;

| expr: NUMBER { $$ = $1; }   
| expr '+' expr [ $$ = $1 + $3; ]
| expr '-' expr [ $$ = $1 - $3; ]
| expr '*' expr [ $$ = $1 * $3; ]
| expr '/' expr [ $$ = $1 / $3; ]
| '(' expr ')' [ $$ = $2; ]
|       ;

%}

int c;
while ((c=getchar()) == ' ' || c == '\t') ;
if (c == EOF) return 0;
if (c == '.' || isdigit(c)) {
  /* number */
  ungetc(c, stdin);
  scanf("%lf", &yylval); /* lexical value */
  return NUMBER; /* lexical type */
}
return c;
}
```
YACC overview

• **YACC converts grammar rules & semantic actions into parsing fcn yyparse()**
  - yyparse parses programs written in that grammar, performs semantic actions as grammatical constructs are recognized

• **semantic actions usually build a parse tree**
  - each node represents a particular syntactic type, children are components

• **code generator walks the tree to generate code**
  - may rewrite tree as part of optimization

• **an interpreter could**
  - run directly from the program (TCL)
  - interpret directly from the tree (AWK, Perl?):
    - at each node, interpret children (recursion), do operation of node itself, return result
  - generate byte code output to run elsewhere (Java)
  - generate internal byte code (Python?, ...)
  - generate C to be compiled later

• **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, interpreter is better
  - on the fly / just in time compilers merge these (e.g., .NET, some Java)

Grammar specified in YACC

• **grammar rules give syntax**
• **the action part of a rule gives semantics**
  - usually used to build a parse tree

  ```
  statement : 
    IF ( expression ) statement 
    create node(IF, expr, stmt, 0) 
    IF ( expression ) statement ELSE statement 
    create node(IF, expr, stmt1, stmt2) 
    WHILE (expression ) statement 
    create node(WHILE, expr, stmt) 
    variable = expression 
    create node(ASSIGN, var, expr) 
  ...
  
  expression : 
    expression + expression 
    expression - expression 
  ...
  ```

• **YACC creates a parser from this**
• **when the parser runs, it creates a parse tree**
• **a compiler walks the tree to generate code**
• **an interpreter walks the tree to execute it**
Excerpt from a real grammar

term:

| term '/' ASGNOP term   { $$ = op2(DIVEQ, $1, $4); } |
| term '+' term          { $$ = op2(ADD, $1, $3); } |
| term '-' term          { $$ = op2(MINUS, $1, $3); } |
| term '*' term          { $$ = op2(MULT, $1, $3); } |
| term '/' term          { $$ = op2(DIVIDE, $1, $3); } |
| term '%' term          { $$ = op2(MOD, $1, $3); } |
| term POWER term        { $$ = op2(POWER, $1, $3); } |
| '-' term %prec UMINUS  { $$ = op1(UMINUS, $2); } |
| '+' term %prec UMINUS  { $$ = $2; } |
| NOT term %prec UMINUS  { $$ = op1(NOT, notnull($2)); } |
| BLTIN '(' patlist ')'

| {$$ = op2(BLTIN, itonp($1), $3); } |

| DEC UNCR var           { $$ = op1(PREDECR, $2); } |
| INC UNCR var           { $$ = op1(PREINCR, $2); } |
| var DECR               { $$ = op1(POSTDECR, $1); } |
| var INCR               { $$ = op1(POSTINCR, $1); } |

stmt:

| while {inloop++;} stmt  { --inloop; $$ = stat2(WHILE, $1, $3); } |
| if stmt else stmt       { $$ = stat3(IF, $1, $2, $4); } |
| if stmt                 { $$ = stat3(IF, $1, $2, NIL); } |
| lbrace stmtlist rbrace  { $$ = $2; } |

while:

WHILE '(' pattern rparen   { $$ = notnull($3); }
Excerpts from a LEX analyzer

"++"  { yylval.i = INCR; RET(INCR); }
"--"  { yylval.i = DECR; RET(DECR); }

([0-9]+(\.[0-9]+)|[eE](?=[+\-][0-9]+))? {
    yylval.cp = setsymtab(yytext, tostring(yytext),
    atof(yytext), CON|NUM, symtab);
    RET(NUMBER); }

while  { RET(WHILE); }
for    { RET(FOR); }
do     { RET(DO); }
if     { RET(IF); }
else   { RET(ELSE); }
return { if (!infunc)
      ERROR "return not in function" SYNTAX;
      RET(RETURN);
    }
  { RET(yylval.i = yytext[0]); /* everything else */ }

The whole process

YACC   Lex (or other)

y.tab.c parser  lex.yy.c analyzer

C compiler

a.out
**AWK implementation**

- source code is about 6000 lines of C and YACC
- compiles (almost) without change on Unix/Linux, Windows, Mac

- parse tree nodes:
  ```c
  typedef struct Node {
      int type; /* ARITH, ... */
      Node *next;
      Node *child[4];
  } Node;
  ```

- leaf nodes (values):
  ```c
  typedef struct Cell {
      int type; /* VAR, FLD, ... */
      Cell *next;
      char *name;
      char *sval; /* string value */
      double fval; /* numeric value */
      int state; /* STR | NUM | ARR ... */
  } Cell;
  ```

**Using Awk for testing RE code**

- regular expression tests are described in a very small 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
  - little languages
  - programs that write programs
  - mechanization
Unit testing

- code that exercises/tests small area of functionality
  - single method, function, ...
- helps make sure that code works and stays working
  - make sure small local things work so can build larger things on top
- very often used in "the real world"
  - e.g., can’t check in code unless has tests and passes them
- often have tools to help write tests, run them automatically
  - e.g., JUnit

```c
struct {
    int yesno; char *re; char *text;
} tests[100] = {
    1, "x", "x",
    0, "x", "y",
    0, 0, 0
};

main() {
    for (int i = 0; tests[i].re != 0; i++) {
        if (match(tests[i].re, tests[i].text) != tests[i].yesno)
            printf("%d failed: %d [%s] [%s]\n", i,
                   tests[i].yesno, tests[i].re, tests[i].text);
    }
}
```

Record keeping

- record of all bug fixes since August 1987

Nov 26, 2009:
- fixed a long-standing issue with when FS takes effect. a change to FS is now noticed immediately for subsequent splits.
- changed the name getline() to awkgetline() to avoid yet another name conflict somewhere.

Feb 11, 2009:
- temporarily for now defined HAS_ISBLANK, since that seems to be the best way through the thicket. isblank arrived in C99, but seems to be arriving at different systems at different times.

Oct 8, 2008:
- fixed typo in b.c that set tmpvec wrongly. no one had ever run into the problem, apparently. thanks to alistair crooks.

Oct 23, 2007:
- minor fix in lib.c: increase inputFS to 100, change malloc for fields to n+1.
- fixed memory fault caused by out of order test in setsval. thanks to david o’brien, freebsd, for both fixes.

Feb 21, 2007:
- fixed quotation in b.c; thanks to Hal Pratt and the Princeton Dante Project.
Lessons

- people use tools in unexpected, perverse ways
  - compiler writing: implementing languages and other tools
  - object language (programs generate Awk)
  - first programming language

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

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

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