AWK

- **a language for pattern scanning and processing**
  - 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 }
  ```

Structure of an AWK program:

- **a sequence of pattern-action statements**

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

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

- **Operation:**

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

- **Usage:**

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

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
- **operators work on strings or numbers**
  - coerce type 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:

```
{ 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

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

END { print NR }          line count

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

$1 > max { max = $1; maxline = $0 }   print longest line
END      { print max, maxline }
```

Awk text formatter

```
#!/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

- **Usual case: array subscripts are integers**
- **Reverse a file:**

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

- **Making an array:**

  `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 limited to integers
- **canonical example: adding up name-value pairs**

Input:

```
pizza    200
beer     100
pizza    500
beer     50
```

Output:

```
pizza    700
beer     150
```

- **program:**

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

Assembler & simulator for toy machine

- **hypothetical RISC machine (tiny SPARC)**
- **10 instructions, 1 accumulator, 1K memory**

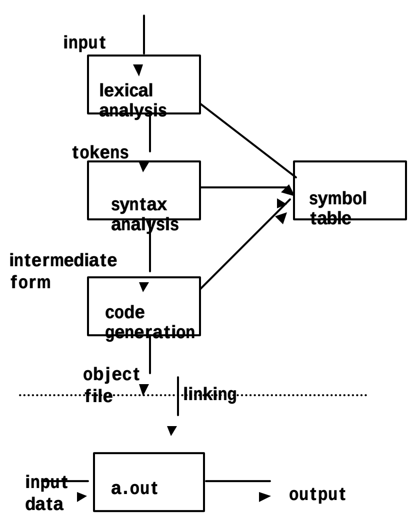
```
# print sum of input numbers (terminated by zero)

     ld    zero   # initialize sum to zero
     st    sum
loop get          # read a number
     jz    done   # no more input if number is zero
     add   sum    # add in accumulated sum
     st    sum    # store new value back in sum
     j     loop   # go back and read another number

done ld    sum    # print sum
     put
     halt

zero const 0
sum  const
```

- **assignment: write an assembler and simulator**

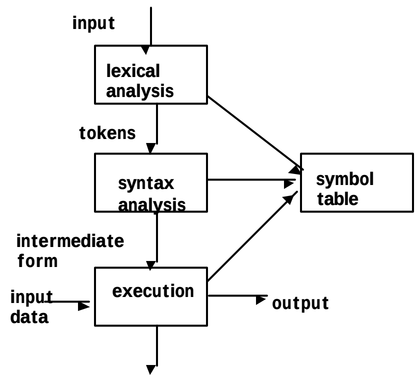
Assembler and simulator/intepreter

```
# asm - assembler and interpreter for simple computer
#   usage: awk -f asm program-file data-files...

BEGIN {
    srcfile = ARGV[1]
    ARGV[1] = ""  # remaining files are data
    tempfile = "asm.temp"
    n = split("const get put ld st add sub jpos jz j halt", x)
    for (i = 1; i <= n; i++)   # create table of op codes
        op[x[i]] = i-1

# ASSEMBLER PASS 1
    FS = "[ \t]+"
    while (getline <srcfile > 0) {
        sub(/#.*/, "")         # strip comments
        symtab[$1] = nextmem   # remember label location
        if ($2 != "") {        # save op, addr if present
            print $2 "\t" $3 >tempfile
            nextmem++
        }
    }
    close(tempfile)

# ASSEMBLER PASS 2
    nextmem = 0
    while (getline <tempfile > 0) {
        if ($2 !~ /^[0-9]*$/)  # if symbolic addr,
            $2 = symtab[$2]    # replace by numeric value
        mem[nextmem++] = 1000 * op[$1] + $2  # pack into word
    }

# INTERPRETER
    for (pc = 0; pc >= 0; ) {
        addr = mem[pc] % 1000
        code = int(mem[pc++] / 1000)
        if      (code == op["get"])  { getline acc }
        else if (code == op["put"])  { print "\t" acc }
        else if (code == op["st"])   { mem[addr] = acc }
        else if (code == op["ld"])   { acc  = mem[addr] }
        else if (code == op["add"])  { acc += mem[addr] }
        else if (code == op["sub"])  { acc -= mem[addr] }
        else if (code == op["jpos"]) { if (acc >  0) pc = addr }
        else if (code == op["jz"])   { if (acc == 0) pc = addr }
        else if (code == op["j"])    { pc = addr }
        else if (code == op["halt"]) { pc = -1 }
        else                         { pc = -1 }
    }
}
```

Anatomy of a compiler



Anatomy of an interpreter



Parsing by recursive descent

```
expr:    term | expr + term | expr -  term
term:    factor | term * factor | term / factor
factor:  NUMBER | ( expr )
```

```
NF > 0 {
    f = 1
    e = expr()
    if (f <= NF) printf("error at %s\n", $f)
    else printf("\t%.8g\n", e)
}
function expr(  e) {      # term | term [+-] term
    e = term()
    while ($f == "+" || $f == "-")
        e = $(f++) == "+" ? e + term() : e - term()
    return e
}
function term(  e) {      # factor | factor [*/] factor
    e = factor()
    while ($f == "*" || $f == "/")
        e = $(f++) == "*" ? e * factor() : e / factor()
    return e
}
function factor(  e) {    # number | (expr)
    if ($f ~ /^[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+)$/) {
        return $(f++)
    } else if ($f == "(") {
        f++
        e = expr()
        if ($(f++) != ")")
            printf("error: missing ) at %s\n", $f)
        return e
    } else {
        printf("error: expected number or ( at %s\n", $f)
        return 0
    }
}
```

YACC and LEX

- **languages for building bigger languages**

- **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

- **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 overview

- **YACC converts grammar rules and semantic actions into a parsing function yyparse()**
- **yyparse parses programs written in that grammar**
- **and performs the semantic actions as grammatical constructs are recognized**

- **yyparse calls yylex each time it needs another input token**
- **yylex returns a token type and stores a token value in an external value for yyparse to find**

- **semantic actions usually build a parse tree**
- **but could just execute on the fly:**

YACC-based calculator

```
%{
#define  YYSTYPE  double   /* data type of yacc stack */
%}
%token   NUMBER
%left    '+' '-'     /* left associative, same precedence */
%left    '*' '/'     /* left assoc., higher precedence */
%%
list:       expr '\n'         { printf("\t%.8g\n", $1); }
   | list expr '\n'    { 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; }
   ;
%%
   /* end of grammar */

#include <stdio.h>
#include <ctype.h>
int       lineno = 1;

main() {  /* calculator */
   yyparse();
}
yylex() {          /* calculator lexical analysis */
   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 */
   }
   if (c == '\n')
          lineno++;
   return c;
}
yyerror(char *s) {  /* called for yacc syntax error */
   fprintf(stderr, "%s near line %d\n", s, lineno);
}
```

YACC overview, continued

- **semantic actions usually build a parse tree**
  - each node represents a particular syntactic type
  - children represent 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
      - do operation of node itself
      - return result to caller
  - generate byte code output to run elsewhere (Java)
    - or other virtual machine instructions
  - generate internal byte code (Perl??, Python?, …)
  - generate C or something else

- **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

Grammar specified in YACC

- **grammar rules give syntax**
- **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**

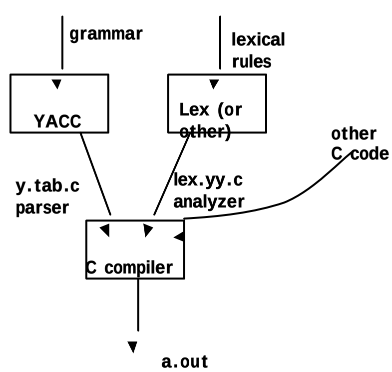
Excerpt from a real grammar

```
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 ')'
             { $$ = op2(BLTIN, itonp($1), $3); }
 | DECR var               { $$ = op1(PREDECR, $2); }
 | INCR var               { $$ = op1(PREINCR, $2); }
 | var DECR               { $$ = op1(POSTDECR, $1); }
 | var INCR               { $$ = op1(POSTINCR, $1); }
```

Excerpts from a LEX analyzer

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

([0-9]+(\.?)[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 */
        }
```

Whole process



AWK implementation

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

- **parse tree nodes:**

```
typedef struct Node {
    int  type;  /* ARITH, … */
    Node *next;
    Node *child[4];
} Node;
```

- **leaf nodes (values):**

```
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;
```

Testing

- **700-1000 tests in regression test suite**
- **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 monaik 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 monaik ghosh for spotting this very subtle one.

- **and some not yet fixed:**

  "Consider the awk program:

  awk '{print $40000000000000}'

  which exhausts memory on the system. this actually occurred in the program:

  awk '{i += $2}
  END {print $i}'

  where the simple typing error crashed the system."

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

Lessons

- **people use tools in unexpected, perverse ways**
  - compiler writing
  - implementing languages, etc.
  - object language
  - 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."
(C. A. R. Hoare, *Hints on Programming Language Design*, 1973)